Ylläpito sähköisen asiointipalvelun ketterässä kehityksessä

Joni Hämäläinen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelytieteiden tutkinto-ohjelma
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Kesäkuu 2016

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joni Hämäläinen: Ylläpito sähköisen asiointipalvelun ketterässä kehityksessä
Pro gradu -tutkielma, 63 sivua, 1 liitesivu
Kesäkuu 2016

Ohjelmiston ylläpitovaiheen aikana muodostuu suurin osa ohjelmiston elinkaaren kuluista. Ylläpitäjät joutuvat päivittäin tasapainottelemaan ohjelmiston ylläpidettävyyden ja asiakkaan muutosvaatimusten välillä. Tässä tutkielmassa pyritään ymmärtämään ylläpidettävyyteen vaikuttavia laatutekijöitä sekä ylläpitoa prosessina. Lisäksi pohditaan ylläpidon roolia ketterässä iteratiivisessa kehityksessä. Ylläpidon, ketterien menetelmien ja palveluliiketoiminnan teoriaa peilataan tapaustutkimuksen kohteena olevaan asiointipalveluun.

Tapaustutkimuksessa suoritetaan kohdeprojektin kehitysprosessista ja ylläpidosta kuvaileva analyysi sekä arvioidaan ylläpidettävyyden ja kehitysprosessin tilaa kyselyn avulla. Tutkimuksen tulos alleviivaa ohjelmiston ylläpidettävyydestä huolehtimisen tärkeyttä. Ylläpidettävyyden täytyy olla korkealla tasolla, kun asiakkaille luodaan arvoa ketterin menetelmin.

Avainsanat ja -sanonnat: ylläpito, ylläpidettävyys, ohjelmiston ymmärrettävyys.

Sisällys

1. Johdanto

Ohjelmistotuotannossa ohjelmiston kehitys on perinteisesti nähty projektina, joka alkaa määrittelyistä ja päättyy valmiin ohjelmiston toimittamiseen asiakkaalle. Käyttöönoton jälkeen ohjelmisto siirtyy ylläpitovaiheeseen. Ylläpitovaiheessa ohjelmistoa muutetaan uusien vaatimusten tai virheiden korjaustarpeiden myötä. Katsaus ohjelmistojen historiaan osoittaa jatkuvien, mutta välttämättömien, muutosten lisäävän ohjelmiston monimutkaisuutta [Lehman, 1980; van Vliet, 2007]. Lisääntyvä monimutkaisuus vaikeuttaa ohjelman ymmärtämistä [April and Abran, 2008], mikä puolestaan laskee ohjelmiston ylläpidettävyyttä. Ajan myötä tästä seuraa kustannuksia, kun uusien muutosten tekeminen vaikeutuu. Monimutkaisuutta voidaan kuitenkin hallita panostamalla ylläpidettävyyteen mahdollisimman aikaisessa vaiheessa ohjelmiston elinkaarta.

Ylläpitoa on kuvailtu jäävuoreksi, sillä moni ylläpidettävyyteen vaikuttava asia on pinnan alla näkymättömissä [Pressman, 2005]. Syy tähän on ylläpidettävyyden mittaamisen haaste: luotettava mittaus on vaikeaa, sillä useat eri ohjelmiston ominaisuudet vaikuttavat ylläpidettävyyteen. Lisäksi ylläpitoon tiukasti kytköksissä oleva ohjelmiston ymmärtäminen on subjektiivinen kokemus. Ylläpidon jäävuorimaisuutta tukee myös fakta, että jos ylläpidettävyyteen liittyviä riskejä ei oteta vakavasti, seuraukset voivat olla erittäin kalliit.

Ylläpidon eri vaiheita on esitetty kirjallisuudessa kattavasti, mutta yhtä oikeaa tapaa hoitaa ohjelmiston ylläpito ja uudistaminen kustannustehokkaasti ei ole vieläkään löydetty. Erilaisia standardeja, malleja sekä metriikoita ylläpitoprosessille löytyy paljon. Tämän tutkielman tavoitteena on kirjallisuutta hyödyntäen ymmärtää ylläpidettävyyteen vaikuttavia laatutekijöitä ja ylläpidon prosesseja. Kirjallisuuskatsauksen avulla on tarkoitus ehdottaa tapoja, joilla sähköisen asiointipalvelun ylläpitoprosessia voidaan tehostaa. Teoriaa tukena käyttäen suoritetaan tapaustutkimus. Tapaustutkimuksen kohteena on Lupapiste-asiointipalvelu, jota on kehitetty neljän vuoden ajan moderneilla ohjelmistotuotannon menetelmillä. Tapaustutkimuksessa Lupapiste-palvelun ylläpitoa analysoidaan pienimuotoisen kyselyn ja tutkielman kirjoittajan omien kokemusten kautta.

Tutkielma jakautuu seuraavasti. Ensin luvussa 2 määritellään termi ylläpidettävyys. Luvussa 3 esitellään ohjelmiston perinteisen ylläpitoprosessin kuvaus ja siihen kuuluvat osa-alueet toimenpiteineen. Luvussa 4 avataan ketterän ohjelmistokehityksen teoriaa ja ketteryyden suhdetta ylläpitoon. Luku 5 käsittelee palveluliiketoiminnan teoriaa, jotta ymmärretään tuotteen ja palvelun ero-

ja sekä palveluun perustuvaa liiketoimintaa. Luvussa 6 suoritetaan tapaustutkimus asiointipalvelun ylläpidosta ja esitetään johtopäätöksiä ylläpitoprosessin ja ylläpidettävyyden parantamiseksi. Lopuksi luvussa 7 tehdään yhteenveto ja loppupäätelmät.

2. Ylläpidettävyys

Ohjelmiston *ylläpidettävyys* (maintainability) on yksi tärkeimmistä luotettavan ohjelmiston ominaisuuksista [ISO/IEC, 2006; Sommerville, 2006]. Sommervillen [2006] mukaan ohjelmisto täytyy rakentaa niin, että sitä voidaan muuttaa asiakkaiden tarpeiden ja toiveiden muuttuessa. Korkea ylläpidettävyys mahdollistaa muutosten teon läpi ohjelmiston elinkaaren.

Kaikkiaan kirjallisuudessa ylläpidettävyys on määritelty hyvin samankaltaisesti: ylläpidettävyys-laatutekijä on tiukasti kytköksissä ohjelmistoon tehtävien muutosten helppouden kanssa. Esimerkiksi Swansonin [1976] mukaan ylläpidettävyys tarkoittaa, että ohjelmaa on helpompaa muuttaa silloin, kun muutostarve on ajankohtainen. ISO/IEC -standardien 9126 [2001] ja 14764 [2006] mukaan ylläpidettävyys taas tarkoittaa kyvykkyyttä muuttaa ohjelmistoa. Pressman [2005] puolestaan kuvailee ylläpidettävyyden helppoudeksi toteuttaa muutoksia ohjelmistoon. ISO/IEC 14764 -standardissa [2006] huomautetaan, että ylläpidettävyys ei kuitenkaan itsessään takaa ohjelman virheiden vähentymistä.

Ylläpidettävyys itsessään koostuu useista laatutekijöistä [Harsu, 2003]. Ohjelmistojen ISO/IEC 9126 -laatustandardin [2001] mukaan ylläpidettävyys koostuu neljästä ohjelmiston ominaisuudesta: *analysoitavuus* (analysability), *muunnettavuus* (changeability), *vakaus* (stability) ja *testattavuus* (testability). Analysoitavuus kuvaa, kuinka vaivatonta ohjelmistosta on tunnistaa virheet. Muunnettavuus kertoo, kuinka helppoa ohjelmistoon on toteuttaa muutoksia. Vakaudella tarkoitetaan odottamattomien haittavaikutusten vähyyttä, kun ohjelmistoa muutetaan. Testattavuus määrittää, kuinka vaivatonta muutetun ohjelmiston toiminnallisuuden varmistaminen on testejä tekemällä. [ISO/IEC, 2001.]

Tuorein ohjelmistojen laatumallien standardi ISO/IEC 25010 [ISO/IEC, 2011] määrittelee ylläpidettävyyden ominaisuudeksi, joka kuvaa ohjelmiston muokattavuutta vaikuttavasti ja kustannustehokkaasti. Standardi määrittelee ylläpidettävyyden koostuvan viidestä ohjelmiston ominaisuudesta: *modulaarisuus* (modularity), *uudelleenkäytettävyys* (reusability), *analysoitavuus* (analyzability), *muokattavuus* (modifiability) ja *testattavuus* (testability). Mainitut ominaisuudet on koottu taulukkoon 1. Tutkielmassa käsitellään ylläpidettävyyttä tuoreemman 25010 -standardin määritelmien mukaan.

Edellä mainittu vanhempi laatustandardi 9126 [ISO/IEC, 2001] jakoi ylläpidettävyyden neljään ominaisuuteen. ISO/IEC 25010 -standardissa säilyivät

edelleen analysoitavuus ja testattavuus, mutta muissa ylläpidettävyyttä määrittelevissä ominaisuuksissa on tapahtunut muutoksia. Chawlan ja Chhbaran [2015] tulkinnan mukaan tuoreemmassa ISO 25010 -standardissa muunnettavuus ja vakaus on yhdistetty uuteen laatutekijään muokattavuus. Lisäksi standardi laajentaa määritelmää ylläpidettävyydestä edellä mainittujen laatutekijöiden lisäksi myös modulaarisuudella ja uudelleenkäytettävyydellä.

Modulaarisuudella tavoitellaan sitä, että ohjelmisto koostuu erillisistä komponenteista [ISO/IEC, 2011]. Kun muutoksia tehdään yhteen komponenttiin, muutosten tarve toisiin komponentteihin on mahdollisimman pieni [ISO/IEC, 2011]. Uudelleenkäytettävyydellä tavoitellaan kompleksisuuden vähentämistä, kun samoja osia voidaan uudelleen käyttää muualla ohjelmistossa [ISO/IEC, 2011]. Muokattavuuteen vaikuttavat sekä modulaarisuus että analysoitavuus [ISO/IEC, 2011]. Modulaarisuus helpottaa muutosten kohdistamista vain haluttuun paikkaan, jolloin minimoidaan riski heijastusvaikutuksista (ripple effects) [Grubb and Takang, 2003]. Analysoitavuus nopeuttaa muutoksen tekijän arviointikykyä järjestelmän tilasta ennen ja jälkeen muutoksen [ISO/IEC, 2011].

Ominaisuus	Selite
Analysoitavuus	Kuinka helppoa ohjelmistosta on tunnistaa virheitä
Modulaarisuus	Ohjelmiston rakentuminen erillisistä, mutta yhdistettävistä komponenteista
Muokattavuus	Kuinka helppoa muutoksia on toteuttaa, ilman odottamattomia haittavaikutuksia
Testattavuus	Testien tekemisen helppous
Uudelleenkäytettävyys	Ohjelmistossa on osia, joita voidaan uudelleen käyttää ja täten vähentää kompleksisuutta

Taulukko 1. Ylläpidettävyyden ominaisuudet [ISO/IEC, 2011].

ISO/IEC 14764 -standardissa [2006] kehotetaan ottamaan ylläpidettävyyden vaatimukset huomioon jo ensimmäisissä ohjelmistoprojektin vaiheissa. Ylläpidettävyyttä tulisi arvioida läpi ohjelmiston kehitysajan. Jos ylläpidettävyyteen panostetaan ajoissa, ylläpitovaiheen kustannukset pienenevät [Bourque and Fairley, 2014; van Vliet, 2007].

2.1. Ylläpidettävyyden ominaisuudet ohjelmistossa

2.1.1. Kompleksisuus

Kompleksisuuden (complexity), eli monimutkaisuuden, voidaan ajatella olevan sekä ohjelmiston ulkoinen että sisäinen ominaisuus. Ulkoisena ominaisuutena eli laatutekijänä kompleksisuus kuvaa, kuinka paljon resursseja tarvitaan ratkaisemaan yksi ongelma. Sisäisenä eli mitattavana ominaisuutena kompleksisuus tarkoittaa lähdekoodin monimutkaisuudesta johtuvia vaikeuksia toteuttaa muutoksia ohjelmistoon [van Vliet, 2007].

Tässä kohdassa keskitytään jälkimmäiseen eli sisäiseen ominaisuuteen, koska ne ovat suoraan mitattavissa, toisin kuin resurssien arviointi. Kuitenkaan ei kannata tehdä liian suoria johtopäätöksiä kompleksisuusmittareiden roolista ylläpidettävyyttä mitattaessa, sillä Sommervillen [2011] mukaan mittareiden suhde laatutekijöihin riippuu aina kehitysprosessista, teknologiavalinnoista ja ohjelmiston sovellusalueesta. Yleisesti ottaen voidaan ajatella, että mitä kompleksisempi ohjelmisto on, sitä vaikeampaa ja kalliimpaa sitä on ylläpitää [Lehman, 1980; Sommerville, 2011].

Kompleksisuus voidaan ajatella joko koko ohjelmiston laajuisena laatutekijänä tai yksittäisten moduulien tai komponenttien monimutkaisuuden mittarina. Esimerkiksi yksittäisen komponentin kompleksisuus todennäköisesti vaikeuttaa ylläpitoa, kun kyseiseen komponenttiin joudutaan tekemään muutoksia [Sommerville, 2011]. Kun ohjelmiston kompleksiset osat tunnistetaan, voidaan resursseja käyttää niiden yksinkertaistamiseen ja siten lisätä ylläpidettävyyttä. Kompleksisten osien tunnistusta varten tarvitaan analyysia ja mittareita.

2.1.2. Modulaarisuus

Ohjelmiston arkkitehtuuri rakentuu yksittäisistä moduuleista (tai komponenteista). Nämä erilliset moduulit yhdistetään ohjelmistossa ratkaisemaan sovellusalueen ongelmia [Pressman, 2005]. Moduulien suunnitteluperiaatteena voidaan käyttää esimerkiksi yhden vastuun periaatetta (single responsibility principle), jolloin yhden moduulin tarkoituksena on ratkaista vain yksi ongelma [Martin, 2002]. Tarkemmin Martin [2002] määrittelee vastuun ”syyksi muuttua” (reason to change), jolloin moduuliin kohdistuvilla muutoksilla täytyy olla vain yksi motiivi. Esimerkiksi jos tulostus-moduuliin tehdään muutoksia, jotka koskevat sekä datan prosessointia että esityskerrosta, niin tällöin moduulilla on enemmän kuin yksi syy muuttua ja se tulisi periaatteen mukaan jakaa kahtia datan prosessoinnin ja esityskerroksen moduuleihin. Periaatetta noudattamalla

saadaan moduulia koskevat muutokset mahdollisimman homogeeniseksi, jolloin on helpompaa ymmärtää moduulin vastuualue. Täten moduulin ylläpito on suoraviivaisempaa. Myös testaus ja virheiden etsintä onnistuu tehokkaammin [Pressman, 2005].

Moduulilla on oma sisäinen olemuksensa sekä ulkoisia yhteyksiä muihin moduuleihin. Moduulin sisäistä tilaa kutsutaan *koheesioksi* (cohesion). Koheesiota kuvaillaan ”liimaksi”, joka pitää moduulin osat yhtenäisinä [Harsu, 2003; van Vliet, 2007]. Korkean koheesion moduuli on mahdollisimman riippumaton muista moduuleista. Moduulin yhteyksiä, eli riippuvuuksia, muihin moduuleihin kuvataan termillä *kytkeytyminen* (coupling). Riippuvuutta voi olla esimerkiksi toisen moduulin toiminnallisuuden (funktioiden/aliohjelmien) tarve tai tiedonvälitys moduulien välillä [Pressman, 2005]. Ylläpidettävyyden kannalta parasta olisi, että moduulit ovat kytkeytyneet toisiinsa mahdollisimman heikosti. Tällöin yhteen moduuliin tehdyt muutokset aiheuttavat vähemmän muutoksia muissa siihen kytkeytyneissä moduuleissa. Näin vähennetään heijastusvaikutusten riskiä [Grubb and Takang, 2003; Pressman, 2005]. Toinen heikon kytkeytymisen etu on, että se mahdollistaa helpommin moduulin uudelleenkäytön [van Vliet, 2007], kun moduulien palveluksia voidaan myöhemmissä vaiheissa valjastaa käyttöön ilman riippuvuuksien tuomia mahdollisia muutostarpeita.

Yhteenvetona voidaan todeta, että yhden vastuun periaatteella moduulien suunnittelussa pyritään korkeaan koheesioon ja heikkoon kytkeytymiseen. Kun moduulien vastuut ovat selkeitä ja keskinäiset riippuvuudet vähäisiä, voidaan ohjelmistoa muokata tehokkaasti. Näin ollen yhden vastuun periaate edistää ohjelmiston ylläpidettävyyttä.

2.1.3. Koodihaju ja tekninen velka

Usein ohjelmistoja kehitetään huomattavan aikataulupaineen alla. Ominaisuudet on tehtävä valmiiksi annetun ajan puitteissa, johon ei välttämättä ole varattu tarpeeksi liikkumavaraa yllätyksiä varten. Tällöin kehittäjälle voi muodostua painetta saada työn alla oleva tehtävä valmiiksi nopeasti. [van Vliet, 2007.] Kiireessä riski ratkaista ongelmat vähemmän laadukkaalla tavalla kasvaa. Jos on tiedossa, että kehittäjä/organisaatio itse ei jatkossa joudu ohjelmistoa ylläpitämään, on riski huonosta koodin laadusta suuri [Sommerville, 2011]. Vähemmän laadukkaat ratkaisut (eli koodihajut) on yleensä kohtuullisen helppo tunnistaa lähdekoodista [van Vliet, 2007].

Koodihaju (code smell) on ohjelmistosta tunnistettava laatupoikkeama. Ensimmäisen kerran koodihajuista puhuivat Fowler ja muut [1999], jotka esitteli-

vät refaktorointimenetelmiä koodihajujen poistamiseksi. Koodihajut viittaavat ongelmiin ohjelmiston laadussa, kuten ymmärrettävyydessä ja muokattavuudessa [Fowler et al., 1999; Yamashita and Moonen, 2012]. Koodihajut ovat siis uhka ohjelmiston ylläpidettävyydelle.

Koodihajujen havainnointi on aina lähdekoodin lukijasta kiinni: kokeneemmat kehittäjät näkevät helpommin rakenteellisia ongelmia koodissa, kun taas vähemmän kokeneet kehittäjät löytävät helpommin koodihajuja lähdekooditasolta [Mäntylä et al., 2004]. Mäntylä ja muut [2004] tekevät mielenkiintoisen havainnon: vähemmän kokeneet kehittäjät tunnistavat lähdekoodista enemmän koodihajuja kuin ohjelmistoa kauemmin kehittäneet kehittäjät. Mäntylä ja muut [2004] arvelevat tämän johtuvan kolmesta mahdollisesta syystä:

1. Kokeneimmille kehittäjille on syntynyt tunneside ohjelmistoon.
2. Lähdekoodi on itse kirjoitettu ja siten tuttua.
3. Koodihajuihin on jo totuttu.

Tutkimuksessa vähiten kokenut kehittäjä tosin oli toiminut kehittäjänä 17 kuukautta, joten käsitettä ”vähemmän kokenut” oli Mäntylän ja muiden mukaan hankala vertailla muiden tutkimusten kanssa.

Koodihajut ja huono arkkitehtuuri johtavat usein teknisen velan ottoon [Measey, 2015]. Teknisen velan otto johtaa järjestelmään, joita on äärimmäisen hankala ylläpitää. Velkaa otetaan usein projektien alkuvaiheessa, jolloin aikataulupaineissa päädytään tekemään nopeita, ja yleensä huonoja, ratkaisuja. Tekniseen velkaan voidaan kuitenkin langeta missä tahansa ohjelmiston elinkaaren vaiheessa. Kuten finanssitalouden veloissa, houkutus ottaa uutta teknistä velkaa ensimmäisen jälkeen on vahva [Measey, 2015]. Velan määrä järjestelmässä saattaa kasvaa erittäin nopeasti ja johtaa nopeasti ylläpitovaikeuksiin [Measey, 2015] ja siten ennakoitua suurempiin kustannuksiin. Velkaa voidaan maksaa takaisin refaktoroimalla [Fowler et al., 1999].

2.2. Ylläpidettävyyden mallit ja metriikat

Vuosikymmenien saatossa ohjelmistoja on mittailtu lukemattomilla eri metriikoilla. Yhtä laatutekijää voidaan mitata valitsemalla sopivat laatutekijää kuvaavat mittarit. Mutta mitattavia ominaisuuksia täytyy olla useita, jotta laatutekijää voidaan arvioida [Harsu, 2003]. Laatutekijän ja mittarin yhteys tulisi ymmärtää, perustella sekä esittää joko kaavalla tai mallilla, jotta mittaustulosta voidaan perustellusti pitää hyödyllisenä laatutekijän kuvaajana [Sommerville, 2011].

Tässä tutkielmassa laatutekijöistä puhuttaessa noudatetaan Harsun [2003] käyttämää terminologiaa. Laatutekijöillä tarkoitetaan ohjelmiston *ulkoisia ominaisuuksia* (external attributes), eli ominaisuuksia, joiden olemus muodostuu useampien yksittäisten ominaisuuksien mittaustuloksista. Ohjelmiston *sisäisiä ominaisuuksia* (internal attributes) kutsutaan Harsun [2003] käytäntöä seuraten yksittäisiksi *mitattaviksi ominaisuuksiksi*. Yksittäinen mitattava ominaisuus voi olla esimerkiksi lähdekoodin rivimäärä (LoC, lines of code). Ylläpidettävyys on laatutekijä, joka koostuu useista mitattavista ominaisuuksista. [Harsu, 2003.]

Ylläpidettävyys on ennuste ohjelmiston ylläpitoon tarvittavasta vaivannäöstä, joka perustuu ohjelmistosta saatuun tietoon [Anda, 2007]. Ylläpidettävyyden mittaaminen yksiselitteisesti on hankalaa, koska ei ole olemassa yhtä ylläpidettävyyden tekijää joka voitaisiin laskea [Grubb and Takanag, 2003; Pressman, 2005]. Ylläpidettävyys laatutekijänä koostuu useista eri mitattavista ominaisuuksista [Harsu, 2003] sekä muista laatutekijöistä [ISO/IEC, 2011]. Lisäksi ylläpidettävyyden voidaan ajatella olevan subjektiivinen kokemus: jokainen kehittäjä tulkitsee lähdekoodia omalla tavallaan, jolloin esimerkiksi kehittäjän kokemuksella voi olla merkittävä vaikutus siihen, kuinka ylläpidettävänä lähdekoodi esiintyy tarkastelijalle.

Ohjelmiston ylläpidettävyyteen on olemassa epäsuoria mittareita. Nämä voidaan jakaa kahteen osaan: ohjelmistotuotteen mittaaminen ja ohjelmistoprosessin mittaaminen. Ensimmäisen tarkoituksena on mitata ohjelmiston ominaisuuksia, jälkimmäisen mittauksella pyritään ymmärtämään, arvioimaan ohjelmistokehitysprosessin menetelmiä ylläpidettävyyden näkökulmasta. Mittaamistapoja ovat joko suora mittaaminen tai epäsuora mittaaminen. Suorassa mittauksessa mittaus suoritetaan vain yhdestä ominaisuudesta, jolloin mittaustulos ei ole riippuvainen muista mittareista. Suoraa mittaustapaa kutsutaan myös ohjelman sisäisten ominaisuuksien mittaamiseksi. Epäsuorassa mittauksessa puolestaan mittaustulos saavutetaan mittaamalla useampia ominaisuuksia, ja tällöin saadaan kuva ohjelmiston ulkoisista ominaisuuksista [Harsu, 2003.].

April ja Abran [2008] toteavat, että vaikka mittareita ja työkaluja ohjelmiston laadun mittaamiseen on olemassa, suorien mittaustulosten spesifisyydestä johtuen niiden tulkitseminen on haastavaa johtotasolla. Tästä johtuen subjektiiviset arviot ovat edelleen suuressa roolissa, kun arvioidaan ohjelmiston ylläpidettävyyttä ja siihen käytettäviä resursseja [April and Abran, 2008]. Andan [2007] tutkimus suosittelee yhdistämään tavalliset ylläpidettävyyteen liittyvät suorat mittarit sekä ammattilaisen subjektiivisen arvion järjestelmän tilasta,

jolloin ylläpidettävyydestä saadaan luotettavampi arvio. Tällaisesta hybridimittauksesta voisi olla hyötyä myös johtotasolle.

Joka tapauksessa teknologian kehitys asettaa haasteita ylläpidettävyyden mittaamiselle, joten ei välttämättä ole perusteltua perustaa laajamittaisia mittausmenetelmiä ylläpidettävyyden arvioimiseksi [Anda, 2007].

2.2.1. Mallit

Chawla ja Chhbara [2015] esittelevät SQMMA (Software Quality Model for Maintainability Analysis) -mallin, jossa matemaattisilla kaavoilla pyritään arvioimaan ylläpidettävyyttä. Mallin mittaukset perustuvat ISO/IEC 9126 ja 25010 -standardeissa esitettyihin ominaisuuksiin (vrt. taulukko 1). Ohjelmiston ominaisuuksiin pohjautuvien mittausten lisäksi Chawla ja Chhbara [2015] mittasivat ohjelmistoprosessista rekisteröityjen muutospyyntöjen avulla virheiden tiheyttä (defect density) sekä virheellisten lähdekooditiedostojen tiheyttä (buggy files density). Näitä epäsuoria mittareita voidaan kutsua hybrideiksi, sillä niissä hyödynnetään sekä ohjelmistosta että prosessista saatavia suoria mittaustuloksia.

Wagey ja muut [2015] ovat kehittäneet mielenkiintoisen ylläpidettävyyden mallin hyödyntäen koodihajumetriikoita. Mallissa suuri määrä suoria ohjelmiston mittareita on yhdistettynä viiteen eri koodihajuun. Nämä koodihajut puolestaan vaikuttavat viiteen ylläpidettävyyden ominaisuuteen (vrt. taulukko 1). Koska yksittäinen ylläpidettävyyden arvo ei kerro juurikaan mitään todellisuudesta, Wagey ja muut [2015] ovat verranneet ylläpidettävyyden mittaustulosta suunnittelumallien tiheyteen kuudessa verrokkiohjelmistoissa. Tuloksena on melko vahva korrelaatio ylläpidettävyysmittarin ja suunnittelumallien tiheyden mittarin välillä.

2.2.2. Ohjelmistotuotteen metriikat

Ohjelmistotuotteesta voidaan tunnistaa useita laatutekijöitä ja mitattavia ominaisuuksia, joilla voidaan nähdä olevan yhteys ylläpidettävyyteen. Kirjallisuudesta löytyy paljon erityisesti olio-ohjelmointiin liittyviä metriikoita [Anda, 2007; Kalkkila, 2011; Yamashita and Moonen, 2012]. Useimmiten ylläpidon metriikat liittyvät ohjelmiston kokoon, modulaarisuuteen, kompleksisuuteen tai koodihajuihin.

Modulaarisuudesta voidaan saada mittaustuloksia esimerkiksi riippuvuusanalyysillä (depedency analysis) [Harsu, 2003], jossa analysoidaan ohjelman rakenteiden välisiä riippuvuuksia. Esimerkki tällaisesta riippuvuudesta on moduulien kytkeytyneisyys. Riippuvuusanalyysin tuloksia voidaan tutkia esi-

merkiksi riippuvuusmatriiseilla (depedency matrix) [Kalkkila, 2011] tai ristiviittauslistauksilla (cross reference listing) [Harsu, 2003].

Eräs tunnettu ja hyvin yksinkertainen kompleksisuusmetriikka on SLoC (source lines of code), joka kuvaa ohjelmiston kokoa lähdekoodin rivien määränä. SLoC on helppo mitata, mutta tarjoaa ainoastaan ymmärrystä ohjelmiston kokoluokasta. Kuitenkin SLoC voi olla luotettava mittari ylläpitohaasteiden ennustamiseen [Sommerville, 2006] esimerkiksi niissä tapauksissa, kun moduulin rivimäärä kasvaa huomattavan suureksi. Toinen tunnettu kompleksisuuden mittari on McCaben syklomaattinen kompleksisuus (cyclomatic complexity), jossa mitataan ohjelmistossa esiintyvien sisäkkäisten kontrollirakenteiden määrää [Harsu, 2003]. McCaben mukaan kontrollirakenteiden määrällä ja sisäkkäisyydellä on vaikutusta ohjelmiston monimutkaisuuteen [Harsu, 2003]. Kolmas tunnettu ja hieman monimutkaisempi mittari on Halsteadin teoria. Teoria perustuu väitteeseen, että algoritmeilla on vähimmäislaajuus. Halsteadin teoriassa ohjelmistoa mitataan laskemalla operaattoreiden ja operandien määrää [Harsu, 2003; Pressman, 2005.] Kompleksisuutta voidaan mitata oletetun vähimmäislaajuuden ja todellisen laajuuden, eli operaattorien ja operandien määrän, suhteena [Harsu, 2003].

Näitä metriikoita ei avata tarkemmin tässä tutkielmassa, sillä niiden oletetaan olevan lukijalle suhteellisen tuttuja. Edellä mainitut metriikat ovat hyvin yleisluontoisia. Lisäksi on selvää, että näiden mittareiden mukaan ohjelmiston kompleksisuus kasvaa ohjelmiston koon kasvaessa. van Vlietin [2007] mukaan suurempi koko ei välttämättä tarkoita sitä, että ohjelmisto olisi kompleksisempi, vaan mittareita tulisi suhteuttaa rivimääriin. Esimerkiksi syklomaattinen kompleksisuus voitaisiin laskea suhteessa ohjelman tai moduulin rivimäärään, tarkastelukulmasta riippuen.

Kompleksisuusmetriikoiden tapauksessa täytyy aina huomioida, että ohjelmistosta saataviin tuloksiin vaikuttavat useat lähdekoodin olomuotoon liittyvät seikat: ohjelmointiparadigma, käytetty ohjelmointikieli sekä ohjelmointityyli. Esimerkiksi olio-ohjelmointiin on kehitetty vuosien saatossa huima määrä erilaisia metriikoita, jotka pyrkivät kuvaamaan olio-ohjelmoinnille tyypillisiä ominaisuuksia [Kalkkila, 2011]. Koska kompleksisuusmetriikat ovat riippuvaisia monesta edellä mainitusta muuttujasta, voidaan van Vlietin [2007] mukaan todeta, että metriikat kertovat *jotain* ohjelmiston kompleksisuudesta. Yhtä ja oikeaa metriikkaa mittaamaan kompleksisuutta ei ole olemassa, vaan kyseessä on aina viitteellinen mittaustulos.

Koodihajuja on mahdollista tunnistaa automaattisesti lähdekoodista [Mäntylä et al., 2004; Yamashita and Moonen, 2012; Yamashita and Counsell, 2013]. Koodihajujen tunnistaminen ja tulkitseminen on helpompaa kuin perinteisten mittareiden [Yamashita and Counsell, 2013], vaikkakin automaattisesti tunnistetut hajut eivät välttämättä korreloi ihmisen arvion kanssa [Mäntylä et al., 2004]. Automaattisen tunnistamisen mahdollisuudesta johtuen koodihajujen valjastaminen ylläpidettävyyden mittaamiseen on houkutteleva ajatus [Yamashita and Moonen, 2012].

Yamashitan ja Moonen [2012] tutkimuksesta ei kuitenkaan löydetty selkeää tapaa mitata koodihajujen ja ylläpidettävyyden suhdetta. Kuten useissa muissakin tutkimuksissa [Anda, 2007] Yamashita ja Moonen päätyivät siihen, että on tarpeen yhdistää useita eri menettelytapoja saadaksemme luotettavampia tuloksia ylläpidettävyyteen vaikuttavista ominaisuuksista.

2.2.3. Ohjelmistoprosessin metriikat

Ylläpidettävyyttä voidaan mitata myös kehitysprosessista. Tällöin kyse on yleensä epäsuorasta mittaamisesta. Kun kehitysprosessia mitataan, voidaan mittauksen ajatella liittyvän ylläpitokulujen arviointiin ja mittaamiseen [Sommerville, 2011; Yamashito and Moosen, 2012].

Yksi epäsuora mittari on MTTC eli mean-time-to-change. Sen avulla voidaan mitata, kuinka kauan korjauksen havaitseminen, toteutus, testaus ja toimitus kestävät [Pressman, 2005.] Näin voidaan epäsuorasti arvioida, kuinka ylläpidettävä ohjelmisto on.

Toinen mahdollinen epäsuora mittari ylläpidettävyydelle on DRE eli defect removal efficiency (virheiden poistotehokkuus). DRE:n voidaan ajatella olevan laadunvarmistuksen mittari. DRE voidaan laskea seuraavasti:

$$DRE = E/(E + D),$$

missä E on virheiden määrä ennen version julkaisua ja D on virheiden määrä seuraavan version jälkeen [Pressman, 2005]. Pressmanin [2005] mukaan DRE:n mittaamisella voidaan rohkaista projektin henkilöstöä löytämään mahdolliset ongelmat jo ennen kuin loppukäyttäjät havaitsevat niitä. DRE on joustava mittari, sillä se voidaan ottaa käyttöön monella eri prosessitasolla. Haasteena DRE:n mittaamisessa on instrumentoida luotettavat mittarit, joilla seurataan virheiden määrää prosessissa. Lisäksi täytyy määrittää, minkä tason virheitä pyritään mittaamaan, onko käytössä koko prosessiskaala vaatimusmäärittelystä testaukseen vai keskitytäänkö vain tiettyyn osaan prosessia. Ylläpidettävyyden mittaamisen näkökulmasta kiinnostavaa olisi esimerkiksi seurata bugi-

raporttien määrää versioittain sekä niiden korjaamiseen käytettyä aikaa (esimerkiksi em. mean-time-to-change mittarilla). Tällöin voidaan arvioida muutosten tekemisen vaikeutta.

DRE:n käyttöönotto voi olla sen joustavuuden takia raskasta, sillä sitä voidaan käyttää hyvin laaja-alaisesti [Kan, 2002; Suma and Nair, 2009] erilaisten prosessin ominaisuuksien mittaamiseen.

Kuten todettu, ylläpidettävyyden mittaus kehitysprosessista liittyy ylläpitokulujen arviointiin. Kirjallisuudesta löytyy edellä mainittujen lisäksi useita malleja ja metriikoita, joita on pyritty hyödyntämään ylläpitotarpeiden arvioinnissa. Yksi käytetyimmistä malleista on COCOMO II [Boehm et al., 2000], jossa työmäärää pyritään arvioimaan ylläpidon osalta muun muassa ohjelmiston koon sekä uudelleenkäytettävyyden määrällä [Roihuvaara, 2013]. COCOMO II on ollut inspiraationa eri arviointimallien kehitykselle, esimerkiksi Leung [2002] esittelee ylläpidon arvioinnin mallin käyttämällä COCOMO:a ja lähimmän naapurin menetelmää.

2.3. Ohjelmiston ymmärtäminen

Tehtäessä muutoksia ohjelmistoon on elintärkeää ymmärtää ohjelma ja sen konteksti, jotta ymmärretään muutoksen vaikutukset ja täten minimoidaan haitalliset heijastusvaikutukset [Grubb and Takan, 2003; van Vliet, 2007]. Ohjelmiston ymmärtäminen (program comprehension) onkin avainasemassa tehokkaan ylläpidettävyyden mahdollistamiseksi, sillä jopa puolet muutokseen vaadittavasta ajasta menee varsinaisen muutoksen toteuttamisen sijasta järjestelmän ymmärtämiseen [Padula, 1993; Grubb and Takang, 2003]. Ennen muutoksen toteuttamista täytyy ensinnäkin ymmärtää, mitä järjestelmä tekee ja miksi ja kuinka se vaikuttaa ympäristöönsä [Grubb and Takang, 2003]. Toiseksi täytyy tunnistaa minne, kaikkialle järjestelmän sisällä muutos vaikuttaa, ja kolmanneksi täytyy olla perusteellinen käsitys siitä, kuinka muutoksen kohteena oleva järjestelmän osa toimii [Grubb and Takang, 2003]. Padula [1993] arvioi, että Hewlett-Packardin insinöörit kuluttivat 200 miljoonan dollarin arvosta aikaa vuodessa vain lähdekoodin lukemiseen ja ymmärtämiseen. Luku on kärjistetty, mutta auttaa hahmottamaan ymmärtämiseen tarvittavien resurssien kokoluokkaa. Cornelissen ja muut [2009] toteavat ymmärtämiseen kuluvan jopa 60% ohjelmistotuotannon työmääristä. Voidaankin todeta, että ohjelmiston ymmärrettävyys on suuri osa ohjelmiston ylläpidettävyyttä, johon panostamalla voidaan synnyttää säästöjä ylläpitovaiheessa.

Ymmärtämisen tavoitteena on selvittää järjestelmän olemus ja syyt sille, miksi järjestelmä on sellainen kuin se on. Ymmärtääkseen järjestelmää tarkastelija tarvitsee tuekseen dokumentoitua tietoa, kuten vaatimusmäärittelyjä, suunnitteludokumentteja ja lähdekoodin. Muiden kuin lähdekoodin kohdalla dokumentaation oikeellisuuden ja ajantasaisuuden säilyttäminen ovat haasteita, sillä niiden tietosisältö ei pysy lähdekoodin tasolla ellei ajantasaisuuteen panosteta. Tämä on tärkeää, sillä dokumentaation ajantasaisuus auttaa ymmärryksen muodostamisessa. Jos dokumentaatio ei ole ajan tasalla, muutoksen tekijän motivaatio etsiä tietoa muualta kuin lähdekoodista hiipuu. [Harsu, 2003; van Vliet, 2007.]

2.3.1. Ymmärtämisen mallit

Ohjelmiston ymmärtämiseen liittyviä prosesseja pyritään kuvaamaan malleilla, jotka perustuvat psykologian perusteisiin [Harsu, 2003]. Ihminen muodostaa kohtaamistaan ilmiöistä (esimerkkinä television toiminta) jatkuvasti mentaaleja malleja, joiden perustana toimivat kognitiiviset prosessit. Näiden mallien perusteella ihminen osaa esimerkiksi ennustaa, mitä tapahtuu jonkin tapahtuman seurauksena (televisio laitetaan päälle, kuva ilmestyy). Kun ihminen saa uutta tietoa ilmiöstä, mentaalimalli päivittyy. Mentaalimallin tarkkuus riippuu ihmisen tiedontarpeista. Esimerkiksi mekaanikon tarvitsee tietää enemmän television toiminnasta kuin tavallisen kuluttajan [Grubb and Takang, 2003.]

Yleisellä tasolla on olemassa kolme ymmärtämisen mallia. Näistä ensimmäinen on *osittava malli* (top-down). Siinä lähdetään liikkeelle sovellusalueen (ohjelmiston toiminnot) ongelmien ymmärryksestä, ja pyritään siten korkeamman tason kautta ymmärtämään matalamman tason toiminnallisuutta. [Harsu, 2003]. *Kokoavassa mallissa* (bottom-up) puolestaan lähdetään liikkeelle ohjelmiston lähdekoodin ymmärtämisestä. Ohjelmoija tunnistaa lähdekoodista samankaltaisia rakenteita. Näitä tunnistettuja rakenteita yhdistelemällä hän hahmottaa korkeamman tason rakenteet ja siten ymmärtää lopulta ohjelmistoa paremmin. [Harsu, 2003.] Kokoavasta mallista voidaan käyttää myös nimitystä *palasteleva malli* (chunking model), joka viittaa ymmärryksen muodostuvan pieni pala kerrallaan kohti suurempaa kokonaisuutta eli korkeampaa abstraktiotasoa [Grubb and Takang, 2003; von Mayrhauser and Vans, 1997]. Kolmas malli on edellä mainittujen yhdistelmä *tilanteen mukainen* (opportunistic) malli, jossa ohjelmoija käyttää edellä mainittuja malleja tilanteen mukaan ohjelmistosta löytyneiden vihjeiden perusteella. [Grubb and Takang, 2003; Harsu, 2003.]

Aloittelevat ohjelmoijat turvautuvat usein kokoavan mallin käyttöön, kun taas kokeneemmat ohjelmoijat käyttävät joko osittavaa mallia tai tilanteen mukaista mallia [Pennington, 1987; Harsu, 2003; Shargabi et al., 2015]. Kokeneimpien ohjelmoijien mentaalimalli sisältää enemmän tietoa ohjelmistosta ja sen kontekstista kuin aloittelijoiden mentaalimallit.

Edellä mainittuja malleja ovat kehittäneet eteenpäin muun muassa Penningtonin ja von Mayrhauser [Shargabi et al., 2015].

Penningtonin malli sisältää kaksi tasoa. Ensin ymmärretään *ohjelman* malli (program model), joka muodostuu kontrollirakenteista ja perusoperaatioista. Toiseksi, kun ohjelman mallin on omaksuttu, ymmärretään *tilanteen* malli (situation model). Tilanteen mallissa hahmotetaan ohjelman toiminnallisuus ja ongelma-alue. Tilanteen malli koostuu tiedonkulun (data flow) ja olennaisten funktioiden hahmottamisesta. [Shargabi et al., 2015]

Von Mayrhauserin malli laajentaa Penningtonin mallia kolmannella eli *sovellusalueen* (domain) tasolla. Ohjelman mallin tasoa korkeampaa tietoa täytyy hankkia manuaalisesti, sillä mikään tekninen apuväline ei pysty analysoimaan lähdekoodia ja tuottamaan tietoa, joka auttaa ymmärtämään sovellusaluetta [von Mayrhauser and Vans, 1997].

2.3.2. Ymmärtämisen osat

Sovellusalueen (problem domain) ymmärtäminen on tärkeä osa-alue, jotta muutoksen perusteet voidaan ymmärtää. Suuret ohjelmistot, esimerkiksi terveydenhuoltojärjestelmät, rakennetaan usein ratkaisemaan tietyn sovellusalueen ongelmia. Koska ohjelmistot usein pilkotaan pienempiin hallittavimpiin osiin (eli moduuleihin), sovellusalueen ymmärrys auttaa hahmottamaan moduulien loogiset suhteet, jolloin on helpompi arvioida muutoksen vaikutukset muihin ohjelmiston osiin. Täten ylläpitäjä voi arvioida muutokseen tarvittavat resurssit luotettavammin. [Grubb and Takang, 2003]

Suoritusvaikutusta (execution effect) arvioidessaan ylläpitäjällä on korkealla abstraktiotasolla ennakkoaavistus siitä, millaisia tulosteita ohjelmisto antaa tietyille syötteille. Kun mennään matalammalle abstraktiotasolle, ylläpitäjä tekee oletuksia siitä, millaisia tuloksia tietty ohjelmiston koodinpätkä tuottaa suorituksen aikana. Suoritusvaikutuksen ymmärrys on mahdollista saavuttaa, kun ylläpitäjä osaa hahmottaa ohjelmiston tietovirtauksen. [Grubb and Takang, 2003]

Syy-seuraussuhteiden (cause-effect relation) avulla voidaan ymmärtää, kuinka eri komponentit keskustelevat keskenään ohjelman suorituksen aikana.

Komponenttien suhteiden hahmotus auttaa arvioimaan muutoksen vaikutuksen laajuutta sekä helpottaa ongelmien etsintää. [Grubb and Takang, 2003]

Tuotteen ympäristön (product environment) ymmärrys on tärkeää ylläpitäjälle, sillä muutokset ympäristössä voivat aiheuttaa suuriakin muutoksia ohjelmistossa. Ympäristön ymmärtäminen helpottaa ohjelman muutostarpeiden arviointia. [Grubb and Takang, 2003]

2.3.3. Ymmärtämisen strategiat

Ymmärtäminen on pohjimmiltaan *iteratiivinen prosessi*, jossa lähtökohtana on epämääräinen hypoteesi ohjelman tilasta [Grubb and Takang, 2003]. Hypoteeseja muodostetaan kaikilla ymmärtämisen abstraktiotasoilla [von Mayrhauser and Vans, 1997]. Tarkastelija tutkii ohjelmistoon liittyvää dokumentaatiota ja lähdekoodia hypoteesia vasten. Tämän jälkeen alkuperäistä hypoteesia jalostetaan saadun tiedon perusteella [Grubb and Takang, 2003].

Penningtonin [1987] mallissa ohjelman tarkastelijat käyvät läpi kaksi vaihetta, kun he yrittävät ymmärtää ohjelmistoa. Ensimmäisessä vaiheessa ymmärrystä muodostetaan ohjelmiston rakenteesta, kuten kontrollirakenteista. Toisessa vaiheessa tietämys laajenee tiedonkulun (data flow) ymmärryksen myötä tilannemalliksi. Tällöin tarkastelija ymmärtää, mitä ohjelma tekee.

Grubb ja Takang [2003] puolestaan jakavat ymmärtämisen iteratiivisen prosessin kolmeen vaiheeseen:

1. Ohjelman tulkitseminen (dokumentaatio, tieto- ja kontrollikaaviot)
2. Lähdekoodin tulkitseminen (järjestelmärakenne, tietorakenteet)
3. Ohjelman ajo (dynaaminen käyttäytyminen, tulosteet).

Vaiheet käytännössä vastaavat osittaisen mallin ymmärryksen vaiheita. Näiden vaiheiden välillä tarvitaan usein iteraatioita, jotta aavistukset ja epäilykset (hypoteesit) saadaan vahvistettua ja siten ymmärretään asiaa paremmin. Grubb ja Takang [2003] huomauttavat, että ensimmäinen vaihe saattaa jäädä kokonaan pois, jos järjestelmään liittyvä dokumentaatio on huonolla tasolla.

Tekstimuotoinen lähdekoodi ja erityisesti sen evoluutio asettavat haasteita ymmärtämiselle [Bourque and Fairley, 2014]. Usein nykyistä koodin tilaa ja sitä edeltäneitä muutoksia ei ole dokumentoitu, vaikka kirjallisuudessa usein tähän rohkaistaan [Kajko-Mattsson, 2008; Sommerville, 2006]. Ei ole myöskään yleensä mahdollista haastatella muutoksen tehnyttä kehittäjää, joka voisi avata lähdekoodin nykyisen tilan taustoja [Pressman, 2005]. Kuitenkin lähdekoodin evoluution tarkastelun tukena voidaan käyttää versionhallintajärjestelmän versiohistoriaa, josta lähdekooditiedostojen muutoksia voi tutkia. Nykyiseen tilaan

johtaneen evoluution ymmärtäminen voi olla mahdotonta, jos tiedostoon tehdään usein muutoksia tai tiedostoa on äskettäin refaktoroitu runsaasti.

2.4. Ylläpidettävyyden edistäminen

Koska ylläpidettävyys on helppoutta toteuttaa muutos ohjelmistoon, ylläpidettävyyttä voidaan edistää mahdollistamalla muutosten toteuttaminen tehokkaasti. Investoimalla ohjelmiston ylläpidettävyyteen vaikuttaviin ominaisuuksiin voidaan ylläpitokulujen olettaa laskevan pidemmällä tähtäimellä. ISO 25010 -standardi [ISO/IEC, 2011] antaa yleisemmällä tasolla viitteitä niistä ohjelmiston elementeistä (vrt. taulukko 1), joihin investoiminen todennäköisesti parantaa ylläpidettävyyttä: analysoitavuus, modulaarisuus, muokattavuus, testattavuus ja uudelleenkäytettävyys. Korkeammalla tasolla voidaan puhua ”teknisestä erinomaisuudesta” (technical excellence), joka Agile-manifestin [Beck et al., 2001] perusteella korostaa hyvää arkkitehtuuria ja teknisen velan välttämistä [Measey, 2015].

Kuten todettu, ohjelmiston ymmärtäminen muodostaa suuren osan muutosten toteuttamiseen tarvittavasta ajankäytöstä. Analysoitavuudella on suuri vaikutus ymmärtämisen tehokkuuteen. Lähdekoodin analysoitavuutta edistävät esimerkiksi sovitut koodityylit ja -käytännöt sekä lähdekoodin yksinkertaisuus ja kommentit [Harsu, 2003].

Modulaarisuutta voidaan parantaa hyvällä arkkitehtuurisuunnittelulla, jossa pyritään yksittäisten moduulien korkeaan koheesioon ja heikkoon kytkeytymiseen. Muokattavuutta voidaan edistää yksinkertaisilla rakenteilla, joilla on mahdollisimman vähän kytköksiä. Näin vältytään mahdollisilta heijastusvaikutuksilta. Muokattavuuteen liittyy täten myös modulaarisuus. Jotta muokkauksia voidaan tehdä luotettavasti, täytyy ohjelmiston testikattavuuden olla hyvällä tasolla. Testikattavuudella pyritään varmistamaan, että ohjelma toimii niin kuin sen on tarkoitettu toimivan.

Kajko-Mattssonin [2008] tutkimuksessa organisaatiot näkivät suurena haasteena järjestelmäymmärryksen, joka väheni useissa tutkimuksen kohteina olleissa organisaatioissa arkkitehtuurin rappeutumisesta johtuen. Syyksi rappeutumiselle nähtiin tehdyt toteutusratkaisut, jotka poikkesivat yleisistä käytännöistä. Tutkimustulos osoittaa tarpeen ohjelmointityylien ja -käytäntöjen määrittämisen projekteissa.

Jotta sovituista käytännöistä on hyötyä, niiden täytyy olla osa laadunvarmistusprosessia (quality assurance). Kokonaisuudessaan ylläpidettävyydestä tulisikin pitää huoli laadunvarmistuksessa. Yksi tehokas tapa on suorittaa kat-

selmointeja (peer review), joissa tuotettua lähdekoodia arvioidaan tarkistuslistan avulla [Sommerville, 2011]. Katselmoinnissa voidaan seurata, ovatko ylläpidettävyyden laatutekijät projektin sopimalla tasolla. Näin ohjelmoijille syntyy positiivnen paine tuottaa laadukasta koodia katselmointiin, jotta katselmoinnin jälkeen aikaa ei tarvitse kuluttaa virheiden korjailuun.

Loppujen lopuksi ylläpidettävyys ja koodin laatu on koko kehitys- ja ylläpitotiimin vastuulla. Jos ylläpidettävyyteen vaikuttavista seikoista ei kollektiivisesti pidetä huolta, riski teknisen velan määrän kasvusta on todellinen. Täten yhteiset ja selkeät pelisäännöt ylläpidettävyyden vaalimiseen ovat avain laadukkaan ja ylläpidettävän ohjelmiston kehityksessä.

2.5. Työkaluna code-maat

Code-maat [Tornhill, 2015] on komentorivityökalu, joka louhii versionhallintajärjestelmän historiatietoja. Koska työkalu perustuu historiatietojen analysoinnille, se on riippumaton käytetystä ohjelmointikielestä. Tornhillin [2015] ajatuksena on tutkia ohjelmistoa sen evoluution (eli historiatiedon) kautta. Tähän versionhallintajärjestelmien historiatiedot antavat oivan mahdollisuuden. Staattisiin analysointityökaluihin verrattuna code-maatin vahvuutena on nimenomaan mahdollisuus tutkia ohjelmiston evoluutiota. Tornhillin [2015] mukaan historiaa tutkimalla saadaan tietoa, jota yksi ohjelmistoversion tila ei kerro.

Ylläpidettävyyden näkökulmasta code-maat tarjoaa mielenkiintoisia metriikoita. Code-maatilla voidaan esimerkiksi tutkia moduulien loogista kytkeytymistä historiatiedon valossa: kun moduulit ovat loogisesti kytkeytyneitä toisiinsa, niihin tehdään usein samaan aikaan muutoksia. Versionhallinnan lokeista saadaan selville, mitkä tiedostot muuttuvat usein yhdessä. Tornhillin mukaan [2015] tämä indikoi moduulien välisestä kytkeytymisestä. Code-maat voi paljastaa yllättävääkin kytkeytyneisyyttä historiatiedon avulla. Näin voi olla esimerkiksi tapauksissa, joissa tiedostoilla ei lähdekooditasolla ole riippuvuuksia toisistaan, mutta niitä kuitenkin muutetaan usein yhdessä. Tämän kaltainen tieto voi auttaa paljastamaan ohjelmiston osia, jotka tarvitsevat uudelleenkirjoitusta tai parempaa jaottelua. Code-maat voi louhia versiohistoriasta monia muitakin ylläpidettävyyden kannalta tärkeitä metriikoita, kuten lähdekoodin ikää (milloin tiedosto viimeksi muuttunut) ja moduulien henkilöriippuvuuksia (kenellä on eniten tietoa moduulista). Lisäksi aikavälianalyysit tarjoavat mielenkiintoista dataa esimerkiksi julkaistujen versioiden väliseen seurantaan. Näin voitaisiin esimerkiksi saada seuraavaksi julkaistavasta versiosta tilastollis-

ta dataa niistä osista ohjelmistoa, joissa tapahtui eniten muutoksia. Näitä osia voitaisiin tarkastella erityisellä huolellisuudella esimerkiksi katselmoinneissa.

3. Ylläpito prosessina

Historiallisesti ylläpito on jäänyt organisaatioissa varsinaisen kehitysprosessin varjoon, kun resursseja panostetaan aikataulupaineiden myötä projektin kehitysvaiheeseen. Tilanteessa on viime vuosina nähty parannuksia, kun organisaatiot pyrkivät maksimoimaan kehitykseen panostamansa rahat satsaamalla aikaisessa vaiheessa ohjelmiston elinkaaren ylläpitoon [Bourque and Fairley, 2014]. Myös avoimen lähdekoodin suosio ja kehitysyhteisöt ohjaavat ohjelmistokehityskulttuuria kohti ylläpidettävämpiä ohjelmistoja [Bourque and Fairley, 2014], kun ohjelmiston täytyy olla laajalle yhteisölle ymmärrettävää ja muutosten teon helppoa.

Alkuun on hyvä tunnistaa projektiluontoisen ohjelmistokehityksen ja pitempiaikaisen ohjelmiston ylläpidon eroja. Ensinnäkin, ohjelmistokehitysprojektin tuloksena on usein *tuote*, kun taas ohjelmiston ylläpidossa asiakkaalle tarjotaan *palvelua* [van Vliet, 2007]. Toiseksi, projekteilla on usein aikataulu, jossa selkeä loppupiste on näköpiirissä. Ylläpidossa päättymispäivää ei ole, vaan asiakkaalta tulevia pyyntöjä ja toiveita toteutetaan liukuhihnalta, usein kiireellä [Heeager and Rose, 2015]. Kolmanneksi organisaation järjestäytymisessä on useimmiten ero. Projektit toteutetaan usein dynaamisempana matriisiorganisaationa, kun ylläpidosta puolestaan vastaa vakaampi ja osastomaisempi organisaatio [Heeager and Rose, 2015].

Grubb ja Takang [2003] arvioivat eri lähteiden perusteella ylläpidon kattavan 40–70% koko ohjelmiston elinkaaren kuluista. Erään lähteen mukaan ylläpidon osuus kuluista on jo 90 % [Seacord et al., 2003]. Voitaneen siis todeta ylläpidon osuuden ohjelmistoprojektien kulurakenteesta olevan merkittävä. Täten aikaisessa vaiheessa ylläpidettävyyteen panostaminen voi tuoda kustannussäästöjä pitemmällä aikavälillä, kun muutoksia voidaan toteuttaa vielä vuosia käyttöönoton jälkeenkin.

Harsu [2003] määrittelee ylläpidon ohjelmistotuotannon vaiheena, joka käynnistyy käyttöönoton jälkeen ja loppuu, kun ohjelma poistetaan käytöstä. ISO/IEC -standardit 12207 [2008] ja 14764 [2006] määrittelevät ohjelmiston ylläpidon prosessiksi, jossa ohjelmistoa muutetaan sen toimituksen jälkeen. Ylläpito ei siis ole vain virheiden korjausta, vaan se kattaa kaikki ohjelmistoon tehtävät muutokset käyttöönoton jälkeen.

ISO/IEC 14764 [ISO/IEC, 2006] on ohjelmiston ylläpitoon keskittyvä standardi. Standardin mukaan ohjelmisto tarvitsee ylläpitoa, jotta voidaan varmistaa, että käyttäjien (muuttuvat) vaatimukset tyydytetään. Standardissa huo-

mautetaan, että ylläpitovaihe on tarpeen aina, oli kehittämismallina käytetty sitten vesiputousmallia tai ketteriä menetelmiä. Nykyisessä ohjelmistokehityksen maailmassa, jossa ohjelmistoja kehitetään usein ketterillä menetelmillä (luku 4), ylläpitovaihe lähenee olemukseltaan varsinaista kehitysvaihetta. Ohjelmiston toimittaminen suurena valmiina pakettina on yhä harvinaisempaa, kun ohjelmistoja kehitetään evoluutionaarisesti iteraatio kerrallaan. Tällöin kehityksen ja ylläpidon erottaminen toisistaan on entistä vaikeampaa [Sommerville, 2011].

Evoluutio eroaa Harsun [2003] mukaan ylläpidosta siten, että evoluutioksi katsotaan kaikki ohjelmiston elinkaaren aikaiset muutokset. van Vlietin [2007] mukaan suurin osa ylläpidosta onkin vain ohjelmiston evoluutiota. Koska ohjelmisto ratkoo reaalimaailman ongelmia, ja reaalimaailma muuttuu jatkuvasti, täytyy myös ohjelmiston muuttua jatkuvasti [van Vliet, 2007]. Ohjelmiston jatkuva muuttumistarve on ymmärretty jo 1970-luvulla, jolloin Lehman [1980] on hahmotellut ohjelmiston evoluution lakeja, joista ensimmäinen sopii yhteen edellisen van Vlietin [2007] toteamuksen kanssa: ohjelmistoa on jatkuvasti muutettava tai siitä tulee käyttäjilleen vähitellen käyttökelvoton [Lehman, 1980].

Vaikka ylläpidon aikana suoritettavat toimet ovatkin lähes samoja kuin kehitysvaiheessa suoritettavat toimet, lähtökohdiltaan ohjelmiston ylläpito on erilaista kuin uuden ohjelmiston kehittäminen. Muutoksia tehdään käytössä olevaan järjestelmään [van Vliet, 2007], jolloin täytyy varmistua, että muutosten jälkeen järjestelmä toimii kuten on tarkoitettu. Lisäksi muutostoiveet ovat kooltaan usein pienempiä kuin kehitysvaiheessa, eikä niiden arviointiin välttämättä tarvita kokeneempien mielipiteitä [April and Abran, 2008].

On huomattava, että ohjelmiston ylläpitoon liittyvä terminologia ei ole kirjallisuudessa yhtenäistä [Sommerville, 2011]. Esimerkiksi Kajko-Mattsson [2001] kritisoi ohjelmiston ylläpidon määritelmää ja taksonomiaa epämääräiseksi ja vaillinaiseksi. Tämän tutkielman kohdassa 3.1 kuitenkin esitetään kirjallisuudessa jo 1980-luvulta alkaen esiintynyt jaottelu ylläpitotoimenpiteistä. Kyseinen jaottelu on löytänyt myös tiensä ISO-standardeihin, joita avataan tarkemmin kohdassa 3.5. Jaottelun yleisyydestä huolimatta Kajko-Mattssonin [2001] esittämälle kritiikille voidaan antaa huomiota, sillä toteutettaessa muutoksia ohjelmistoon joskus voi olla epäselvää, mihin ylläpitokategoriaan mikäkin ohjelmiston muutos kuuluu.

Tässä luvussa tutustutaan perinteisiin ylläpitoprosessin vaiheisiin ja niihin vaikuttaviin seikkoihin. Tarkoituksena on avata ohjelmiston ylläpidon histori-

allista taustaa ja roolia ohjelmiston elinkaaressa. Ylläpidon nykytilaa ketterien menetelmien rinnalla pohditaan tarkemmin kohdassa 4.7.

3.1. Ylläpidon tarve

Muutoksen tarve johtaa ohjelmiston ylläpidon tarpeeseen. Jatkuvista ohjelmistoon kohdistuvista muutospaineista on tiedetty jo vuosikymmeniä sitten. Lehmanin [1980] laeista ensimmäisen mukaan ohjelmasta tulee ajan mittaan käyttäjilleen vähemmän hyödyllinen, ellei ohjelma uudistu jatkuvasti.

Grubb ja Takang [2003] listaavat ylläpidon motivaattoreita:

- Jatkuva palvelu: kriittisten ohjelmistojen toimintakäyttö täytyy varmistaa, joten ylläpitoa tarvitaan esimerkiksi virheiden korjaamiseen ja komponenttipäivitysten mahdollistamiseen.
- Pakolliset muutokset: muutokset esimerkiksi lainsäädännössä altistavat ohjelmiston muutostarpeille.
- Käyttäjäkokemuksen parantaminen: käyttäjiltä tulevien parannusehdotusten toteuttaminen on ylläpitovaiheessa usein tarpeen.
- Tulevaisuuden ylläpitotyön helpottaminen: ohjelmiston rakenteen tai dokumentaation parantaminen on usein taloudellisesti järkevää, jotta ikääntyvän ohjelmiston muutostarpeiden toteuttaminen on mahdollista.

Nämä motivaattorit kattavat hyvin muussakin kirjallisuudessa esitettyjä syitä ylläpidon tarpeelle [van Vliet, 2007; Bourque and Fairley, 2014].

Johtoportaan näkökulmasta ylläpito voidaan helposti ajatella lisäkuluna, joka olisi pitänyt välttää tekemällä ohjelmistosta laadukkaampi ja oikeampi jo kehitysvaiheessa. Deklava [1992] kuitenkin osoittaa, että laadukkaasti ja moderneilla menetelmillä tehdyllä ohjelmistolla ylläpitokulut ovat alkuvaiheen jälkeen suuremmat kuin heikompi laatuisen ohjelmiston. Selitys tälle ilmiölle on Deklavan mukaan ylläpidettävyydessä: hyvin tehtyä ohjelmaa on helpompi ylläpitää ja siihen voidaan lisätä kohtuullisen vaivattomasti uusia ominaisuuksia vastaamaan loppuasiakkaiden tarpeisiin. Silti luvussa 2 esiin tuotu väite ylläpitokulujen pienenemisestä ylläpidettävyyteen panostamisella pitää edelleen paikkaansa myös Deklavan tutkimuksen valossa, sillä laadukkaasti toteutetun projektin ylläpitokustannukset olivat ensimmäisen vuoden ajan pienemmät kuin verrokkiprojekteissa. Lisäksi moderneilla tekniikoilla toteutetussa ohjelmistossa virheiden korjaamiseen käytettiin vähemmän aikaan verrattuna

vanhempien menetelmien ohjelmistoihin [Deklava, 1992], jolloin painopiste on ollut asiakkaiden palvelimisessa uusien toiminnallisuuksien muodossa.

Ylläpidon tarve alkaa lisääntyä ajan myötä, kun käyttäjät alkavat ymmärtämään järjestelmää paremmin, jolloin he alkavat tunnistamaan puuttuvia ominaisuuksia [van Vliet, 2007]. Tästä loppukäyttäjien ymmärryksen muodostumisesta muutostoiveita alkaa kerääntyä vasta jonkin aikaa järjestelmän käyttöönoton jälkeen. Ylläpitoa ei tulisi nähdä kulueränä, vaan investointina asiakkaan palvelemiseen.

3.2. Ylläpitotoimenpiteet

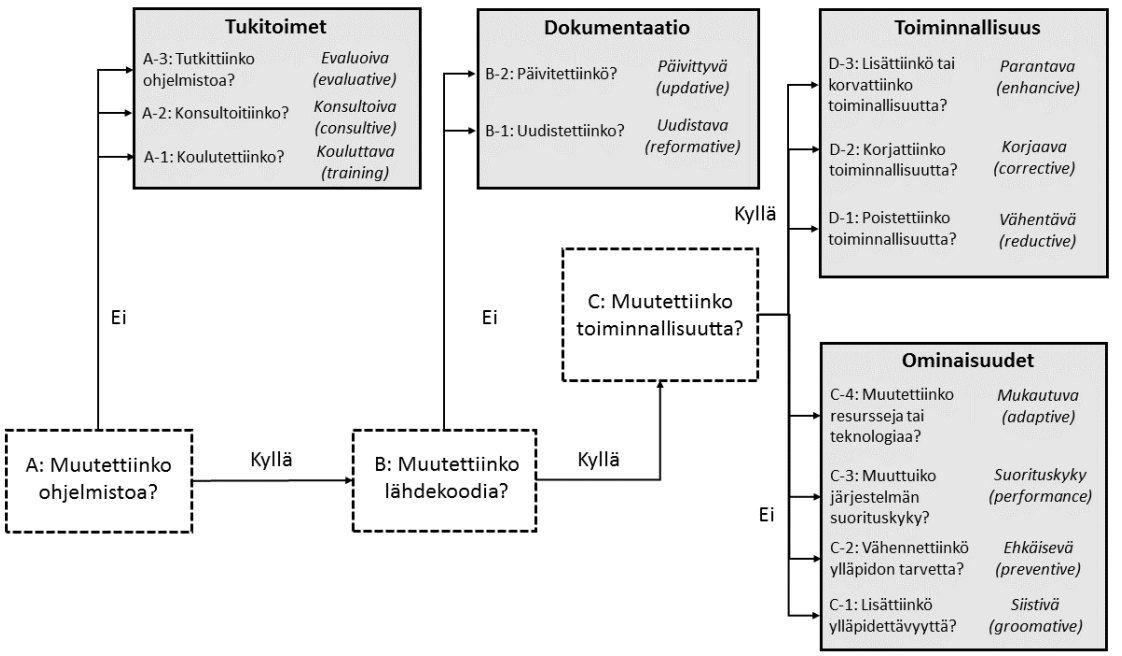
Ylläpitoprosessin näkökulmasta yksi ensimmäisistä ylläpidollisten toimenpiteiden luokittelijoista oli Swanson [1976]. Hän jakaa ylläpitotoimet kolmeen luokkaan: *korjaavaan* (corrective), *mukautuvaan* (adaptive) ja *täydellistävään* (perfective) ylläpitoon. Ensimmäiseen luokkaan kuuluvat esimerkiksi ohjelman prosessointivirheet (”bugit”), suorituskykyongelmat ja toteutuspuutteet. Mukautuvaan ylläpitoon Swanson luokittelee kuuluvaksi ohjelman muutokset, joilla mukaudutaan muuttuvaan dataan (esimerkiksi tietokannan uudelleenjärjestely) tai muuttuvaan ympäristöön (uusi laitteisto). Viimeiseen luokkaan Swansonin mukaan kuuluvat toimet, joilla ei välttämättä korjata olemassa olevia vikoja tai puutteita, vaan esimerkiksi parannetaan yleistä suorituskykyä tai parannetaan ohjelmiston ylläpidettävyyttä.

Näiden lisäksi myöhemmin on luokitteluun lisätty myös neljäs käsite *ehkäisevä* (preventive) ylläpito [Lientz and Swanson, 1980; ISO/IEC, 2001] erottamaan toiminnalliset parannukset (täydellistävä ylläpito) ja ohjelmiston ylläpidettävyyteen vaikuttavat muutokset toisistaan. Swansonin [1976] ensimmäisissä määritelmissä ehkäisevä ylläpito kuului täydellistävän ylläpidon luokkaan. Ehkäisevän ylläpidon tavoitteena on usein ennaltaehkäistä suurempia (ylläpidollisia) ongelmia myöhemmissä ohjelmiston elinkaaren vaiheissa. Ehkäiseviä toimia voi olla esimerkiksi ohjelmiston rakenteen parantaminen tai dokumentoinnin lisääminen [Grubb and Takang, 2003]. Ehkäisevä ylläpito voidaan siis nähdä investointina ylläpidettävyyteen. Valitettavasti ehkäisevää ylläpitoa tehdään yleensä vain, kun hyödyt ovat suuremmat kuin kustannukset [Swanson, 1976]. Ohjelmiston rakenteen parantelun investointia on hankala perustella, koska muutoksia ei voi nähdä, sillä tarkoituksena ei ole lisätä suorituskykyä tai toiminnallisuutta vaan panostaa rakenteeseen ja ylläpidettävyyteen.

Vaikka edellä esitellyt luokat esiteltiin ylläpitotoimenpiteiden luokkina, kirjallisuudessa käytetään myös samoja luokkia luokittelemaan *muutosta* (change).

Luokittelut ylläpitotoimenpiteiden ja muutosten välillä voidaankin nähdä synonyymeina. Esimerkiksi termit ”korjaava ylläpito” ja ”korjaava muutos” tarkoittavat samaa: virheellisen toiminnan havaitsemisen seurauksena ohjelmistoa ylläpidetään toteuttamalla korjaava muutos.

Ylläpitotoimenpiteiden jaottelusta ei kuitenkaan olla yhtä mieltä. Kuten Kajko-Mattsson [2001] myös Chapin ja muut [2001] kritisoivat ylläpidon termistöä. Chapin ja muut [2001] huomauttavat, että eri tutkijat ovat käyttäneet samoja termejä tarkoittamaan eri asioita, jolloin termien merkitys kirjallisuudessa on hämärtynyt. Erityisesti hämärtyneet ovat Swansonin [1976] ensiksi esittämät ylläpitotoimenpiteiden termit korjaava (corrective), mukautuva (adaptive) ja täydellistävä (perfective). Lisäksi IEEE [1990] on sanastossaan määrittänyt samat termit, mutta Chapinin ja muiden [2001] mielestä IEEE:n määritelmät ovat osittain ristiriidassa Swansonin määritelmien kanssa. IEEE:n [1998] ylläpitostandardi määrittelee Chapinin ja muiden [2001] mielestä jälleen samat termit hieman poikkeavasti Swansonin [1976] ja IEEE sanaston [1990] määritelmistä. Chapin ja muut [2001] huomauttavat lisäksi, että osa tutkijoista on käyttänyt ISO/IEC -standardeissa 12207 ja 14764 esiintyviä termien määritelmiä, jotka myös poikkeavat edellä mainittujen osapuolten määrityksistä. Ylläpitotoimenpiteiden termistön käyttö on siis kirjallisuudessa hyvin pirstoutunut.

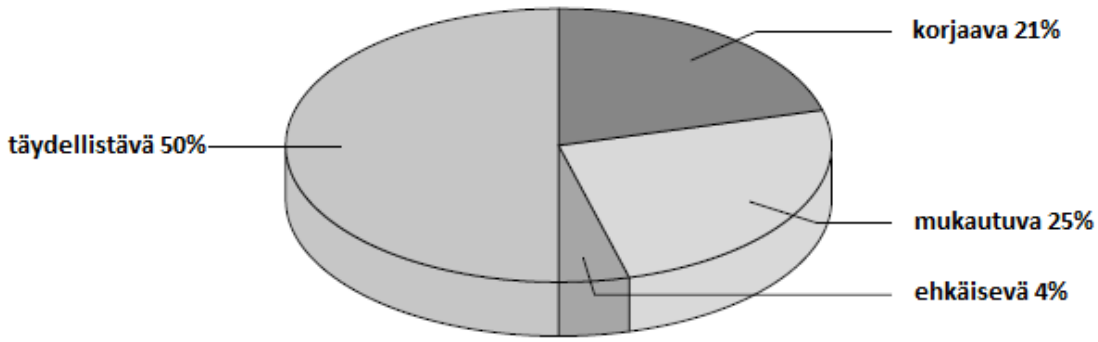


Kuva 1. Chapin ja muiden [2001] esittämä päätöspuu ylläpitotyypin valintaan.

Chapin ja muut [2001] esittävät vaihtoehtoisen luokittelun ohjelmiston ylläpidolle ja samalla sen evoluutiolle. Sen sijaan, että ohjelmiston ylläpito jaetaan

edellä mainittuihin luokkiin, Chapin ja muut esittävät hienojakoisemman luokittelun, jossa toimenpiteet jakautuvat ensin neljään klusteriin (kuvan 1 harmaat laatikot): 1) *toiminnallisuus* (business rules), 2) *ohjelmiston ominaisuudet* (software properties), 3) *dokumentaatio* ja 4) *tukitoiminnot* (support interface). Näiden sisällä on määritetty tarkemmat tyypit, jotka ovat nähtävissä kuvan 1 päätöspuusta. Päätöspuun avulla voidaan jaotella suoritetut ylläpitotoimenpiteet niiden lopputuloksen, ei etukäteen luokittelun, perusteella. Kuvan 1 mukaisesti voidaan nähdä Chapinin ja muiden [2001] tunnistavan 12 varsinaista ylläpitotoimenpidettä. Näillä Chapin ja muut [2001] pyrkivät hienojakoisempaan jaotteluun, jonka termit ovat käytettävissä niin ylläpidon toteuttajille, esimiehille kuin tutkijoillekin.

Ylläpito mielletään usein virheiden korjaamiseksi. On kuitenkin huomattava, että ylläpitotehtäviin käytettävästä ajasta yli puolet kuluu muuhun kuin varsinaisiin korjaaviin toimenpiteisiin [Lientz and Swanson, 1980; Pressman, 2005].



Kuva 2. Ylläpitotoimien jakautuminen [van Vliet, 2007].

Kuva 2 osoittaa, että vain reilu 20 % ylläpitoon käytettävästä ajasta on virheiden korjausta (korjaavaa ylläpitoa). Kuvan Kuva 2 data on peräisin Lientzin ja Swansonin [1980] tutkimuksista. Samasta kuvasta voidaan nähdä, että 75% ylläpidosta on joko täydellistävää tai mukautuvaa ylläpitoa, eli käytännössä uuden kehitystä [Ahmad et al., 2016]. van Vlietin [2007] mukaan ylläpitotoimien jakautumisessa ei ole tapahtunut suurta muutosta vuosikymmenien aikana, vaan eri ylläpitotoimiin käytettävä aika jakautuu edelleen kutakuinkin samoin kuin kuvassa 2.

Grubb ja Takang [2003] summaavat ylläpitoon vaadittavat toimenpiteet ylläpitäjän näkökulmasta seuraavasti:

1. Ylläpitäjä hankkii ymmärryksen nykyisestä järjestelmästä ja sen kontekstista.
2. Muutos ohjelmistoon tehdään.
3. Uusi versio testataan ja julkaistaan.

Näiden lisäksi ennen toteutusta tarvitaan perustellut päätökset, että muutokselle on todellakin tarve ja se on mahdollinen toteuttaa [Grubb and Takang, 2003].

van Vliet [2007] jakaa puolestaan ylläpitäjän ylläpitotehtävät seuraaviin vaiheisiin:

1. *Eristämisvaiheessa* (isolation) selvitetään mikä osa ohjelmistosta täytyy muuttaa.
2. *Muutos* (modification) toteutetaan muuttamalla yhtä tai useampaa komponenttia.
3. Järjestelmä *testataan* (testing) muutoksen jälkeen, jotta voidaan varmistaa ohjelmiston toimivan kuten on oletettu.

van Vlietin [2007] mukaan eristämisvaihe vie noin 40% ylläpitoon käytettävästä ajasta, kun muutos- ja testausvaihe kummatkin vievät 30%.

3.3. ISO/IEC/IEEE -ohjelmistojen ylläpitoon liittyvät standardit

Suuret standardiorganisaatiot ISO, IEC ja IEEE ovat vuosikymmenien saatossa julkaisseet useita standardeja ohjelmistokehitykseen liittyen. ISO/IEC/IEEE 12207-2008 -standardi on erittäin laaja standardi, joka kattaa ohjelmiston elinkaaren prosessit (software life cycle processes). ISO/IEC 14764-2006 -standardi on laajennus vanhemmasta ISO/IEC 12207 (vuoden 1995) -standardista. ISO/IEC 14764 -standardissa keskitytään ohjelmiston ylläpitoprosessin määrittelemiseen.

Edellä mainittujen standardien mukaan ohjelmiston ylläpitoprosessin tarkoituksena on mahdollistaa kustannustehokkaat tukitoimet toimitetulle ohjelmistotuotteelle [ISO/IEEE 2006, 2008].

ISO/IEC 12207 -standardi [ISO/IEC, 2008] listaa onnistuneen ylläpitoprosessin tuloksia:

- Muutosten hallintaan on laadittu ylläpitostrategia, joka noudattelee ohjelmiston julkaisustrategiaa.
- Muutosten vaikutukset olemassa olevaan järjestelmään on tunnistettu.

- Järjestelmä- ja ohjelmistodokumentaatiota päivitetään tarvittaessa.
- Toteutetuille muutoksille kirjoitetaan asianmukaiset testit, jolloin varmistetaan jo olemassa olevien vaatimusten täyttyminen muutosten jälkeenkin.
- Päivitykset saatetaan asiakkaan ympäristöön toimivana.
- Ohjelmiston muutokset kommunikoidaan kaikille tarvittaville osapuolille.

Korkeamman tason ISO/IEC 12207 -standardi jakaa ohjelmiston ylläpidolliset toimet viiteen osaan. *Prosessin käyttöönottovaiheessa* (process implementation) varmistetaan ylläpitoprosessin suunnitelmat ja menetelmät, jotta ylläpidon eri tehtävät voidaan toteuttaa tehokkaasti. Menetelmiin sisältyy ohjelmiston käyttäjiltä kerättävät ongelmaraportit ja muutospyynnöt sekä käyttäjien suuntaan toimiva palautemekanismi. Ylläpitoprosessissa tulee myös olla määriteltynä, kuinka muutokset tuodaan hallitusti nykyiseen järjestelmään.

Ongelman ja muutoksen analysointivaiheessa (problem and modification analysis) ylläpitäjä arvioi muutospyynnön tyypin, laajuuden ja kriittisyyden. Ongelman tai muutospyynnön tarve täytyy todentaa järjestelmästä. Analyysin perusteella ylläpitäjä dokumentoi ongelman tai muutostarpeen ja esittää vaihtoehtoja toteutukseksi.

Muutoksen toteutusvaiheessa (modification implementation) ylläpitäjä dokumentoi, mitkä ohjelmiston osat tarvitsevat muutoksia. Ennen toteutusta määritellään testit ja arvioinnit, jotka sekä nykyisen että muutoksen jälkeisen järjestelmän täytyy läpäistä. Toteutuksen jälkeen arvioidaan onko muutos täyttänyt muutosvaatimukset, ja onko järjestelmän muut vaatimukset edelleen täytetty.

Ylläpidon hyväksyntävaiheessa (maintenance review/acceptance) muutos katselmoidaan järjestelmän eheyden takaamiseksi. Lopuksi muutokselle haetaan sopimuksen mukainen hyväksyntä (esimerkiksi asiakkaalta).

Muutoksen käyttöön saattamisessa (migration) suunnitellaan muutoksen käyttöönotto yhdessä asiakkaan kanssa, perustellaan ja esitellään tarve uuden version julkaisulle tai uuteen ympäristöön siirtymiselle. Jos muutokset edellyttävät koulutusta, vanhaa ja uutta järjestelmää saatetaan käyttää rinnakkain muutosten läpiviennin helpottamiseksi. Muutoksen käyttöönoton jälkeen arvioidaan muutoksen onnistuminen ja tiedotetaan asianosaisia osapuolia arvioinnin tuloksista.

Näiden viiden prosessin lisäksi ylläpitoprosessit tarkemmin kuvaavassa ISO/IEC 14764 -standardissa [2006] määritellään viimeiseksi prosessivaiheeksi

ohjelmiston käytöstäpoisto (retirement), mutta tämän toimen käsittely tässä tutkielmassa ei ole tarkoituksenmukaista. [ISO/IEC, 2008; ISO/IEC, 2006.]

Termi	Selitys
Ylläpidettävyys (maintainability)	Ohjelmistotuotteen kyvykkyys muutokseen. Muutoksella tarkoitetaan esimerkiksi korjauksia, parannuksia tai mukautuvuutta ympäristöön, vaatimuksiin ja toiminnallisiin (funktionaalisiin) vaatimuksiin.
Ohjelmiston ylläpito (software maintenance)	Tarvittavat toimet, joilla mahdollistetaan kustannustehokkaat tukitoiminnot ohjelmistolle. Sisältää toimet ennen sekä jälkeen ohjelmiston toimituksen.
Mukautuva ylläpito (adaptive maintenance)	Tavoitteena on pitää ohjelmistotuote käytettävänä muuttuneessa tai muuttuvassa ympäristössä.
Kehittävä ylläpito (perfective maintenance)	Puutteiden tunnistaminen ja korjaus ennen kuin ne ilmenevät häiriöinä (esimerkiksi dokumentaation parannus, suorituskyvyn tiedonkeruu).
Ehkäisevä ylläpito (preventive maintenance)	Mahdollisten virheiden tunnistaminen ja korjaus, ennen niiden havaitsemista käytössä.
Korjaava ylläpito (corrective maintenance)	Korjaa tuotteessa havaittuja ongelmia ja täyttää (muuttuneet) vaatimukset.
Hätäylläpito (emergency maintenance)	Suunnittelematon väliaikainen muutos, jolle myöhemmin toteutetaan korjaava muutos.

Taulukko 2. ISO/IEC 14764 -standardissa määriteltyjä ylläpitoon liittyviä termejä.

IEEE 982.1-1988 -standardissa on esitetty ohjelmiston kypsyysindeksi Software Maturity Index (SMI). Indeksi mittaa ohjelmistojulkaisun vakautta laskemalla, kuinka paljon muutoksia uuteen versioon on tullut edellisen versiojulkaisun jälkeen [IEEE 1988; Harsu, 2003]. IEEE on kuitenkin vuonna 2005 julkaistussa 982.1-standardin laajennoksen liitteissä ehdottanut SMI:n poistoa standardista, koska indeksi ei suoraan mittaa kypsyyttä, vaan moduulien muutosten tiheyttä. SMI:n maksimiarvo on 1.0, joka saavutetaan, kun edelliseen julkaisuun verrattuna uuteen julkaisuun ei tehdä yhtään muutosta. Täten voidaan kyseenalaistaa arvon 1.0 tavoittelu: onko järkevää olla tekemättä mitään, jos ohjelmistoon kuitenkin kohdistuu muutosvaatimuksia? Tuoreessa kirjallisuudessa SMI:in ei enää törmää, joten voitaneen olettaa, että SMI ei ole ollut viime vuosina käytössä organisaatioissa.

IEEE 982.1-1988 -standardi on suunniteltu kriittisten järjestelmien (mission critical systems) kehittämisen standardiksi, joten standardissa suositeltujen mittareiden käyttö ei välttämättä ole perusteltua kevyemmissä ohjelmistoprojekteissa, kuten web-sovellusten kehittämisessä.

Standardeissa määritellyt prosessit ovat erittäin tarkkoja ja kattavat kaikki tällä hetkellä parhaiksi nähdyt toimet laadun varmistamiseksi. Kaikkien prosessivaiheiden toteuttamisesta ja toteutumisen seurannasta aiheutuisi paljon raskasta prosessibyrokratiaa, joten kaikkien vaiheiden käyttöönotto organisaatiossa on tuskin tarpeellista. Työlästä olisi esimerkiksi toteuttaa eri prosessivaiheiden mittausta, sillä standardeissa ei oteta kantaa siihen, miten organisaatioiden kannattaa mitata prosessejaan. Standardit toimivat hyvänä tarkistuslistana toimivan ja laadukkaan ylläpitoprosessin muodostamiseksi organisaatiossa.

3.4. Regressiotestaus

Ylläpidon kannalta testaus ja erityisesti regressiotestaus ovat elintärkeitä vaiheita, joilla varmistetaan ohjelmiston toiminnallisuus myös muutosten jälkeen. Järjestelmän täydellinen testaus on usein mahdotonta aika- ja kustannusrajoitteista johtuen. [IEEE, 1998; Bourque and Farley, 2014]

Regressiotestauksella pyritään varmistamaan järjestelmän toiminta myös ylläpidollisten muutosten jälkeen. Jos järjestelmästä löytyy odottamaton virhe, korjauksen lisäksi luodaan regressiotesti, jolla varmistetaan, ettei sama bugi enää toistu ohjelmassa [Grubb and Takang, 2003]. Muutosten jälkeen kaikki järjestelmään liittyvät testit on syytä ajaa. Näin voidaan varmistua, ettei muutos ole aiheuttanut heijastusvaikutuksia. Kirjallisuudessa kaikkien testien ajaminen on koettu vaivalloiseksi [Sommerville, 2011; van Vliet, 2007], jolloin mahdollisuutena on valita regressiotestausvaiheeseen vain tietyt testit [van Vliet, 2007]. Edellä mainitussa kirjallisuudessa lähtökohtana on ollut manuaalinen testaus. Kuten Sommerville [2011] toteaakin, viimevuosina automaattitestaus on kerännyt suosiota.

Ketterässä Extreme Programming -menetelmässä [Beck, 1999] korostetaan *jatkuvaa integrointia* (continuous integration). Jatkuvan integroinnin tavoitteena on säännöllisin väliajoin varmistua siitä, että ohjelmisto on toimiva kaikkien kehittäjien muutosten jälkeen [Beck, 1999]. Jatkuvaan integrointiin kuuluu osana myös integraatiotestaus, jossa yksikkötestien lisäksi järjestelmän eri komponenttien välisiä integraatioita testataan [Beck, 1999; Sommerville, 2011]. Vuosien aikana jatkuva integrointi on kehittynyt siihen pisteeseen, että testejä voidaan ajaa jatkuvan integraation palvelimella automaattisesti. Automaattitestaus

vähentää testaukseen käytettävää aikaa dramaattisesti, jolloin voidaan keskittää resursseja esimerkiksi testikattavuuden laajentamiseen.

3.5. Dokumentointi

Kohta 2.3 käsitteli ohjelmiston ymmärtämistä. Kuten todettu, dokumentointi on merkittävä osa ohjelmiston ymmärtämistä. Ylläpitäjille dokumentaatio on usein ensimmäinen kosketus järjestelmään [Grubb and Takang, 2003]. Ylläpitoa tehdään, koska järjestelmään tarvitaan muutoksia. Näitä muutostarpeita dokumentoidaan useimmiten *muutospyyntödokumenteilla* (change request) [Harsu, 2003]. Muutospyyntödokumenttien on tarkoitus perustella muutoksen tarve, esimerkiksi esittelemällä ongelma järjestelmässä. Muutospyyntödokumentille voidaan ongelmakuvauksen lisäksi tallentaa esimerkiksi työmääräarvioita, ehdotus toteutuksesta sekä toteutuksen jälkeen kuvaus ratkaisusta [Harsu, 2003]. Kun myöhemmässä vaiheessa toinen ylläpitäjä on tekemisissä koodin kanssa, hän voi kääntyä muutospyyntödokumentin puoleen selvittääkseen syytä nykyiselle koodin tilalle. Tällainen tieto voi olla arvokasta järjestelmän ymmärryksen kannalta. Tämän takia muutospyyntödokumentin sisällön tulee olla laadukasta [Grubb and Takang, 2003], kuten myös kaiken muun ohjelmiston dokumentaation.

De Souzan ja muiden [2006] mukaan ylläpitäjät pitivät lähdekoodin kommentteja tärkeimpänä dokumentaation lähteenä. Tutkimuksessa haastateltiin kehittäjien lisäksi myös analyytikkoja, konsultteja ja päälliköitä, joten otanta on heterogeeninen. Lähdekoodin kommentoinnin tärkeys on luontevaa, koska muutokset tehdään lähdekoodiin, jolloin kommentit ovat nopeasti saatavilla ja voivat kuvata ratkaisua tarkemmalla tavalla kuin muut dokumentaatiotyypit. Muiksi tärkeiksi dokumentaatiotyypeiksi tutkimuksessa [de Souza et al., 2006] nousivat erilaiset tietomallikaaviot sekä vaatimusmäärittelyt.

Kajko-Mattsson [2005] on selvittänyt kyselyssään organisaatioiden korjaavan ylläpidon dokumentaatiotapoja. Suurin osa kyselyyn vastanneista organisaatioista laiminlöi dokumentaation. Vain muutama organisaatio lähes täytti tutkimuksen asettamat dokumentointitavoitteet. Organisaatioilla ei ollut selkeää kuvaa, millainen on hyvin dokumentoitu järjestelmä. Organisaatioiden prosessit eivät tukeneet ylläpitäjien motivaatiota tuottaa dokumentaatiota, sillä dokumentaation ohjeistusta ei ollut sisällytetty ylläpitoprosesseihin. Kyselyyn vastanneet organisaatiot toivoivat dokumentointiin standardia, jonka avulla dokumentaation voisi liittää helposti nykyisiin prosesseihin. [Kajko-Mattsson, 2005]

3.6. Ylläpidolliset ongelmat

Lehmanin lait [1980] eivät imartele jatkuvista muutoksista johtuvia seurauksia ohjelmistoon: kompleksisuus kasvaa, lähdekoodin koko kasvaa ja laatu rappeutuu. Tämä hankaloittaa myöhempien muutosten tekoa ohjelmistoon. Ylläpitäjät joutuvat usein muodostamaan ymmärryksensä suoraan lähdekoodista. Kun koodin laatu huononee, on sitä yhtä vaikeampi ymmärtää. Usein ylläpitäjä joutuu analysoimaan jonkun toisen kirjoittamaa koodia, joka saattaa hankaloittaa ymmärtämistä lisää [Harsu, 2003].

Useat lähteet viittaavat ylläpidon motivaatio-ongelmiin, missä ylläpito nähdään vähempiarvoisena ja epämiellyttävänä tehtävänä [Harsu, 2003; van Vliet, 2007: Sommerville, 2011; Bourque and Fairley, 2014]. Sommerville [2011] kuitenkin viittaa tutkimuksiin, joissa tämä on osoitettu ainakin osittain kulttuurien välisiksi eroiksi. Sommervillen esimerkissä Japanissa arvostetaan ylläpito huomattavasti korkeammalle kuin Yhdysvalloissa.

Ylläpidon ongelmat jaetaan yleisesti ottaen kolmeen [April and Abran, 2008] tai neljään [Bourque and Fairley, 2014] kategoriaan. April ja Abran [2008] jakavat ylläpidolliset ongelmat teknisiin ongelmiin, prosessiongelmiin sekä organisaation tavoitteiden tuomiin ongelmiin. Bourque ja Fairley [2014] puolestaan jakavat ongelmat seuraaviin kategorioihin: tekniset ongelmat, johdon ongelmat, kuluarviointiongelmat sekä mittausongelmat. Jaottelun yksinkertaistamiseksi tässä tutkielmassa ylläpidon ongelmat jaetaan teknisiin ongelmiin ja johdollisiin ongelmiin.

3.6.1. Tekniset ongelmat

Teknisten ongelmien voidaan katsoa kokonaisuudessaan muodostuvan ohjelmiston huonosta ylläpidettävyydestä. Tämän voidaan ajatella liittyvän Lehmanin [1980] lakiin, jossa muutosten myötä lähdekoodin laatu rapistuu. Kuitenkaan lähdekoodin rapistuminen ei saisi olla itseisarvo, vaan ylläpidettävyyttä tulisi vaalia ohjelmiston alkuhetkistä saakka, jotta myöhemmin vältytään teknisiltä ongelmilta.

van Vliet [2007] mainitsee tekniseksi ongelmaksi huonorakenteisen koodin. Huonorakenteista koodia on vaikeampi muuttaa ja riski heijastusvaikutuksista kasvaa [Grubb and Takang, 2003]. Huonorakenteisen koodin ongelma voitaisiin välttää, jos ehkäisevään ylläpitoon panostettaisiin. Kuten todettu, valitettavasti ehkäisevää ylläpitoa tehdään yleensä vain, kun sen hyödyt ovat suuremmat kuin kustannukset, sillä ehkäisevän ylläpidon lisäarvoa loppukäyttäjille ei yleisesti tunnisteta.

Klassinen ylläpidon tekninen ongelma ovat vanhentuneet teknologiat ja arkkitehtuurit [April and Abran, 2008]. Tällöin järjestelmää on erittäin hankalaa ylläpitää. Vaihtoehtoina on pyrkiä tuomaan mukaan uudempaa teknologiaa ja parempaa arkkitehtuuria. Toinen vaihtoehto on korvata vanha järjestelmä kokonaan uudella, mikä on luonnollisesti kallis vaihtoehto.

Kun ohjelmisto on ylläpitovaiheessa, sen teknisiin ongelmiin ei voi enää vaikuttaa painottamalla laadun tärkeyttä kehitysvaiheessa. Tapoja ratkaista teknisiä ongelmia ylläpitovaiheessa kuitenkin on, joskin ne ovat kalliita. Teknisiä ongelmia voi ratkaista ehkäisevän ylläpidon keinoin parantamalla koodin ja dokumentaation laatua [Harsu, 2003; van Vliet, 2007]. Muutokset ohjelmiston rakenteessa ja dokumentaatiossa eivät tuo välittömästi lisäarvoa loppukäyttäjille, mutta ehkäisevä ylläpito tulisikin nähdä pidempiaikaisena investointina laatuun ja palveluun, josta loppukäyttäjä hyötyy pidemmällä aikavälillä.

3.6.2. Johdolliset ongelmat

Kun ylläpitovaihe alkaa intensiivisen kehitysvaiheen jälkeen, on ylläpitäjillä usein edessään suuri määrä tehtäviä, joita ei ehditty toteuttaa varsinaisen kehitysvaiheen aikana [April and Abran, 2008]. Tällaisessa tilanteessa taustalla on kiireessä tehty projekti, jolloin kiire on usein havaittavissa lähdekoodin laadusta. Nämä seikat ovat omiaan laskemaan ylläpitäjän motivaatiota [April and Abran, 2008].

Projektiin panostaminen kehitysvaiheessa on yleistä. Kun ylläpitoa sitten suoritetaan vähemmän laadukkaaseen lähdekoodiin, kulut nousevat. Ylemmälle johdolle ylläpito näyttäytyy äkkiä suurena kulueränä, josta ei ole selkeää hyötyä organisaatiolle. Tässä tilanteessa ylläpidon on vaikea asettua linjaan organisaation tavoitteiden kanssa. [Bourque and Fairley, 2014.]

Sekavan kehitysvaiheen jälkeen dokumentaation puuttuminen ja ajantasaisuus on haaste. Ylläpitäjät eivät luota dokumentaatioon vaan olettavat sen olevan vanhentunutta [van Vliet, 2007], jolloin aletaan vain tulkitsemaan lähdekoodia ilman suurempaa ymmärrystä järjestelmästä. Myös muutospyyntödokumentaation tarkkuus on ongelma, sillä usein dokumentoidaan muutoksen lopputulos, ei lopputulokseen johtaneita perusteluja. [van Vliet, 2007; April and Abran, 2008].

4. Ketterät menetelmät ja ylläpito

Edellisessä luvussa esitelty ylläpitoprosessi ja siihen kuuluvat osat ovat muotoutuneet vuosikymmenien ajan. Vanhahtavat ja vesiputousmaiset tarkasti määritellyt vaiheet ovat vahvasti läsnä ylläpitoa käsittelevässä kirjallisuudessa. Jotta voidaan ymmärtää ylläpitoa tämän päivän ohjelmistoprojekteissa, on välttämätöntä tutustua ketteriin menetelmiin ja niiden taustalla olevaan ideologiaan. Tässä luvussa esitetään ensin ketterät menetelmät pääpiirteittäin. Lopuksi pohditaan ylläpidon suhdetta ketteriin menetelmiin.

4.1. Ketteryydestä yleisesti

Vesiputousmallin jäykkyys rohkaisi kehittämään joustavimpia ohjelmistokehitysprosesseja. Evoluutionaariset kehitysprosessit kuten prototyypitys ja spiraalimalli [Boehm, 1986] tarjosivat vaihtoehtoa vesiputousmallille. Näiden mallien perustana on pyrkimys sopeutua ohjelmistojen evoluutioon. Mallit ovat iteratiivisia: niissä pyritään toistuvien vaiheiden avulla etenemään kohti haluttua lopputulosta, joka on laadukas ohjelmisto [Pressman, 2005; van Vliet, 2007; Sommerville, 2008]. Vuosien saatossa ohjelmistoteollisuudessa on pyritty löytämään tasapaino kehityksen, projektisuunnittelun ja asiakasvaatimusten välillä. Tuloksena on useita eri ketteriä menetelmiä, joilla pyritään vastaamaan ohjelmistoteollisuuden haasteisiin parhailla mahdollisilla tavoilla.

Ketterien menetelmien (agile methodologies) taustalla on ”ketterä liike” (agile movement). Ketterän liikkeen perusperiaatteet on esitelty tunnetussa Agile-manifestissa [Beck et al., 2001]. Ryhmä ohjelmistoalan ihmisiä kokoontui pohtimaan vaihtoehtoa raskaille dokumentaatioon painottuneille ohjelmistokehityksen prosesseille. Tapaamisen tuloksena syntyi manifesti, jonka neljä perusarvoa ovat [Beck et al., 2001]:

- Yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja.
- Toimiva ohjelmisto ennen kattavaa dokumentaatiota.
- Asiakasyhteistyö ennen sopimusneuvotteluja.
- Muutoksiin reagoimista enemmän kuin suunnitelmissa pitäytymistä.

Ensimmäinen arvo korostaa yhteistyötä: ohjelmistokehittäjät ovat yksilöitä, mutta keskustelu ja kanssakäynti johtavat parempaan lopputulokseen kuin tiukasti määritellyt menetelmät ja työkalut. Toisen arvon taustalla on laadukkaan ja ymmärrettävän koodin tuottaminen: kun ohjelmistoa on helppo ymmärtää, se vähentää herkästi vanhenevan dokumentaation tarvetta. Tällöin voidaan

keskittyä tuottamaan laadukasta koodia raskaan dokumentoinnin sijaan. Kolmas arvo viittaa asiakaskeskeisyyteen, sillä onnistunut ohjelmistoprojekti on riippuvainen tyytyväisestä ja innokkaasta asiakkaasta. Kun ohjelmistoa kehitetään asiakkaalle asiakkaan kanssa, ohjelmistosta tulee asiakasvaatimusten mukainen ja asiakas tuntee tuotteen omakseen. Neljännen arvon mukaan muutoksiin täytyy aina valmistautua sekä toteuttaa niitä tarpeen vaatiessa. Neljännen arvon kohdalla voidaan myös löytää synergiaa Lehmanin [1980] ensimmäisen lain kanssa: ohjelmiston täytyy ajan kuluessa muuttua, tai siitä tulee käyttäjilleen hyödytön.

Näiden neljän arvon lisäksi Beck ja muut [2001] ovat avanneet myös kaksitoista periaatetta Agile-manifestin taustalla. Tässä tutkielmassa näitä periaatteita ei tarkastella tarkemmin, sillä edellä mainitut arvot ovat riittävät tämän tutkielman laajuuteen, ja ne myös kattavat ajatusmaailmaltaan Beck ja muiden [2001] tarkemmin kuvaamat periaatteet.

Agile-manifesti voidaan ajatella poliittisena manifestina: se on vastine raskaalle vuosikymmeniä käytetylle vesiputousmallille, jossa ohjelmistokehityksen eri vaiheet suoritetaan vaihe vaiheelta [Pressman, 2005]. Näiden vaiheiden jälkeen tuloksena pitäisi olla toimiva ohjelmisto, jonka voi luovuttaa asiakkaalle. Näitä vaiheita sitovat usein jäykät sopimukset. Vaatimukset määritellään projektin alussa ja arkkitehtuuri suunnitellaan ennen toteutusta. Toteutus on perinteisesti tehty edellisten vaiheiden perusteella. Ei ole jäykän mallin ja sopimusten mukaista, että suunnitelmia muutetaan. Vesiputousmallin tuloksena on usein jäykkiä ohjelmistoja, joiden toiminnallisuus on usein muuta kuin asiakas kuvitteli. Ongelmien korjaus jälkikäteen ylläpitovaiheessa osoittautuu usein erittäin kalliiksi, kun muutoksiin ei ole varauduttu.

Ketterät periaatteet korostavat ihmiskeskeisyyttä prosessien sijaan [Pressman, 2005]. Ihmisten välinen kommunikointi on avainasemassa onnistuneen ohjelmiston luonnissa. Tarkoituksena on nopeasti tuottaa toimivaa ohjelmistoa asiakkaalle, jotta voidaan saada heti palautetta ja tehdä palautteen avulla taas paranneltu versio. Näin voidaan vastata muuttuviin liiketoiminnallisiin vaatimuksiin. Täten periaatteet korostavat myös muutosvalmiutta: muutos on väistämätöntä [van Vliet, 2007; Lehman, 1980].

Agile-manifesti sisältää ketterien menetelmien perusarvot, mutta se ei itsessään sisällä menetelmiä, joilla näitä arvoja voidaan ohjelmistoprojekteissa soveltaa. Projekteihin täytyy sisällyttää kehitysprosessi, jossa ketteriä arvoja voidaan käyttää niin, että projektitiimi voi karsia pois turhat vaiheet ja keskittyä toimittamaan mahdollisimman nopeasti toimivaa ohjelmistoa asiakkaalle

[Pressman, 2005]. Ketterän ohjelmistokehityksen ajatuksia ja periaatteita oli jo olemassa ennen Agile-manifestoa [Takeuchi and Nonaka, 1986; Schwaber, 1995]. Näistä varhaisin realisointi varsinaiseksi menetelmäksi oli Kent Beckin *Extreme Programming* (XP) vuonna 1999. Seuraavaksi esitellään lyhyesti nykyisin keskeiset ketterät menetelmät: XP, Scrum, Lean/Kanban sekä start-upien suosima Lean start-up.

4.2. Extreme Programming (XP)

Yksi ensimmäisistä menetelmistä oli *Extreme Programming* (eli *XP*) [Beck, 1999]. XP keskittyy muutamiin hyviksi havaittuihin ohjelmistokehityksen käytäntöihin ja vie ne nimensä mukaisesti äärimmäisyyksiin. Esimerkki äärimmäisyydestä on XP:n *pariohjelmointi* (pair programming), jossa kaksi kehittäjää toimivat samalla työpisteellä: toinen kirjoittaa koodia ja toinen katselee vierestä. Vanha hyväksi havaittu käytäntö pariohjelmoinnissa on, kun joku muu kuin koodin kirjoittaja katsoo kirjoitetun lähdekoodin läpi. Lähdekoodin läpikäyntiä kutsutaan *katselmoinniksi* (code review). Katselmoinnilla voidaan löytää mahdollisia virheitä ja valvoa koodin laatua verrattuna tilanteeseen, jossa kukaan muu kuin koodin kirjoittaja ei tarkistaisi koodia. XP:ssä tämän käytännön äärimmäisyyteen vienti toteutuu kun yhdelle tietokoneelle resursoidaan jatkuvasti kaksi henkilöä. Tästä työtavasta seuraa myös yksi XP:n perusperiaatteista: laadukas työnjälki.

Muita XP:n vaalimia periaatteita ovat testaus, arkkitehtuuri, yksinkertaisuus ja lyhyet iteraatiot [Beck, 1999]. Testauksen äärimmäisyyksiin vienti toteutetaan kaikkien toiminnallisuuksien yksikkötestauksella. Myös asiakas suorittaa omat testauksensa. Lisäksi integraatiotestaukset suoritetaan päivittäin. Arkkitehtuuria vaalitaan jatkuvasti kaikkien kehittäjien toimesta suunnittelemalla ja rakenteiden parantamisella eli refaktoroimalla (refactoring). Yksinkertaisuuteen XP:ssä toteutuu toteuttamalla toiminnallisuudet niin yksinkertaisesti kuin mahdollista. XP:ssä pyritään tekemään muutoksia vähittäin (incremental change) iteroimalla. Iteraatiot pyritään pitämään mahdollisimman lyhyinä, päiväkin voi olla liian pitkä aika. Kun tehtävä on saatu tehtyä ja yksikkötestit menevät läpi, koodi integroidaan nopeasti (continuous integration) muiden kanssa, jonka jälkeen ajetaan integraatiotestejä koko ohjelmiston toiminnallisuuden varmistamiseksi. Jatkuva integrointi varmistaa, että koko kehitystiimi työstää aina viimeisintä versiota ohjelmistosta. [Beck, 1999.]

Extreme Programming on toiminut 2000-luvulla ketterien menetelmien suunnannäyttäjänä. XP:n käytäntöjä on sovellettu erilaisiin menetelmiin tarpei-

den mukaan. Esimerkiksi viime vuosina jatkuva integrointi (continuous integration, CI) on otettu osaksi organisaatioiden ohjelmistokehitysprosessia, vaikka varsinaista kehitystä ei tehdäkään täysin XP:n käytäntöjen mukaisesti. Extreme Programmingin äärimmäisyyksiin viedyt menetelmät voivat olla joillekin projekteille liian äärimmäisiä. Esimerkki tällaisesta projektista on turvajärjestelmä, jossa vaatimukset ovat tiukkoja ja esimerkiksi testauskäytännöt tarkkaan määriteltyjä turvajärjestelmän laatutekijöiden varmistamiseksi [Sommerville, 2006].

4.3. Scrum

Scrum on viitekehys (framework), jolla ketterää ideologiaa voidaan harjoittaa ohjelmistoprojekteissa. Ensimmäisiä viitteitä Scrumin kaltaisesta kehitystavasta esittävät Takeuchi ja Nonaka [1986] artikkelissa, josta Sutherland [1995] ja Schwaber [1995] saivat idean ottaa käyttöönsä Takeuchin ja Nonakan kuvaama prosessi. Schwaber ja Beedle [2001] kehittivät Scrumin käytäntöjä eteenpäin ja julkaisivat kirjan, jossa menetelmä esitetään.

Scrum keskittyy enemmän varsinaiseen iteratiiviseen kehitysprosessiin kuin kehityksen tekniseen puoleen [Sommerville, 2011]. Scrumin perustana on empirismi, jossa tietämys kasvaa kokemuksen ja faktoihin perustuneiden päätöksien myötä. Scrum rohkaisee tekemään tuotteeseen liittyviä päätöksiä tiiminä, eikä Scrumissa käytetä sanaa projektipäällikkö [Sommerville, 2011]. Tämän näkökulman voidaan nähdä selkeästi tukevan Agile-manifestin ideaa, jossa yksilöiden välistä kommunikointia korostetaan.

Scrumin voidaan ajatella koostuvan *tiimistä* (scrum team), *tapahtumista* (scrum events) ja *artefakteista* (scrum artifacts) [Schwaber and Sutherland, 2013]. Scrumin tapauksessa puhutaan usein *inkrementistä* (increment), jolla käytännössä tarkoitetaan tuotteesta julkaistua versiota. Scrum on inkrementaalinen kehitysprosessi (incremental process), jossa tavoitteena on julkaista uusi versio säännöllisin väliajoin.

Tiimi

Scrum-tiimi on projektin sisällä itseohjautuva, eli päätökset kehitysprosessin suhteen tehdään tiimin sisällä. Scrum-tiimin sisällä on erilaisia rooleja.

Tuoteomistaja (product owner) vastaa kehitettävästä tuotteesta ja on vastuussa tuotetta koskevissa päätöksissä, esimerkiksi siitä, millainen on tuotteen kehityssuunta. Tuoteomistaja on vastuussa niin kutsutusta *backlogista*, joka on ohjelmistoprojektin tehtäväjono, siis tulevat toteutettavat ominaisuudet. Tuote-

omistaja varmistaa, että *kehitystiimillä* (development team) on tekemistä, ja että tekeminen on mahdollisimman tehokasta. [Schwaber and Sutherland, 2013]

Kehitystiimin jäsenevät toteuttavat *inkrementin* (increment). Inkrementin tuloksena tuotteesta julkaistaan uusi versio, joka sisältää uutta toiminnallisuutta. Kehitystiimin pitää olla sopivan kokoinen, jotta inkrementit voidaan suorittaa tehokkaasti. Liian pieni tiimi ei välttämättä saa vaadittua inkrementtiä tehdyksi ja saattaa sisältää tiedon sekä taidon puutteita. Liian suuren tiimin koordinointi on hankalaa. [Schwaber and Sutherland, 2013]

Scrum master on vastuussa Scrumin ymmärtämisestä ja toimeenpanosta. Scrum master toimii laadunvarmistajana ja palvelee sekä tuoteomistajaa että kehitystiimiä [Schwaber and Sutherland, 2013]. Tuoteomistajan suuntaan Scrum master auttaa backlogin hallinnassa ja toimii konsulttina kehitystiimin suuntaan. Scrum master varmistaa, että kehitystiimiä ei häiritä tiimin ulkopuolelta [Sommerville, 2011].

Tapahtumat

Scrumiin kuuluu oleellisena erilaiset tapahtumat (events). Näistä ensimmäinen on *sprintti* (sprint). Sprintti on kiinteä ajanjakso, yleensä 2-4 viikkoa, jonka aikana inkrementti toteutetaan. Sprintti päätyy usein uuden version julkaisemiseen tuotteesta, ellei sprinttiä ole lykätty tai peruttu jonkin ongelman johdosta [Schwaber and Sutherland, 2013].

Sprintin *suunnittelupalaverissa* (sprint planning) koko Scrum-tiimi suunnittelee tulevan sprintin ja valitsee backlogista toteutettavat tehtäväkohdat. Toteutettavat toiminnallisuudet arvioidaan ja sprintti suunnitellaan siten, että kaikkien sprintille otettujen tehtävien tulisi valmistua yhden sprintin aikana [Schwaber and Sutherland, 2013]. Täten tiimi sitoutuu toteuttamaan valitsemansa tehtävät sprintin aikana.

Päivittäinen Scrum-palaveri eli *daily scrum* on lyhyt (5-15 minuuttia) kehitystiimin palaveri, jossa tiimin jäsenet käyvät läpi, mitä edellisen daily scrumin jälkeen on tehty, mitä on tarkoituksena tehdä seuraavaan daily scrumiin mennessä sekä tuodaan esiin mahdollisia ongelmia. Daily scrumin tarkoituksena on parantaa tiimin kommunikaatiota ja levittää tietoa sprintin tilasta sekä mahdollisesti kohdatuista ongelmista. [Schwaber and Sutherland, 2013]

Sprintin *katselmointi* (sprint review) pidetään sprintin lopussa. Se on vapaamuotoinen tapaaminen, jossa kerätään palautetta inkrementistä. Katselmoinnissa käydään läpi, miltä sprintin aikana syntynyt inkrementti näyttää sekä keskustellaan, mitä seuraavassa sprintissä tulisi tehdä [Schwaber and Sut-

herland, 2013]. Varsinainen sprintti kuitenkin suunnitellaan muodollisemmassa sprintin suunnittelupalaverissa.

Ennen uuden sprintin suunnittelupalaveria Scrum-tiimi järjestää *retrospektiivin*, jossa käydään läpi edellinen sprintti kokonaisuutena. Retrospektiivissä voidaan avoimesti keskustella siitä, miten ihmiset, yhteydenpito, prosessi ja työkalut toimivat. Keskustelujen tarkoituksena on oppia ongelmista ja kehittyä kohti parempaa tiimiä ja Scrum-prosessia. Retrospektiivi ei kuitenkaan ole vain ongelmista keskustelua, vaan tarkoituksena on tuoda esille myös edellisessä sprintissä hyvin menneet asiat. [Schwaber and Sutherland, 2013]

Artefaktit

Artefaktien tarkoituksena on Scrumissa tuoda näkyviin kaikki tarpeellinen tieto kaikille Scrum-tiimiläisille [Schwaber and Sutherland, 2013].

Tuotteen työlista eli *backlog* (product backlog) on lista kaikista tuotteen osista, joita *saatetaan* tarvita tuotteen kehityksessä. Jotkut tuotteen työlistalla olevat kohdat ovat tuotenäkökulmasta vähemmän tärkeitä kuin toiset, jolloin työlistaan syntyy prioriteettijärjestys. Tuoteomistaja priorisoi ja yhdessä Scrum masterin kanssa hallitsee listaa. Tuotteen työlistan kullekin kohdalle (esimerkiksi toteutettavalle ominaisuudelle) luodaan kuvaus (perustelut tarpeelle, valmiin vaatimukset), sekä arvioidaan kehitystiimin kanssa, kuinka paljon työtä kohta tarvitsee toteutuakseen. Koska liiketoiminnalliset vaatimukset muuttuvat jatkuvasti, myös tuotteen työlista elää jatkuvasti. [Schwaber and Sutherland, 2013]

Sprintin työlista (sprint backlog) on sprintin suunnittelupalaverissa muodostettu lista käynnissä olevan sprintin tehtävistä kohdista, jotka on sitouduttu tekemään sprintin aikana. Sprintin työlistasta on vastuussa kehitystiimi. Sprintin työlista kuvaa parhaillaan käynnissä olevaa työtä. Scrumin mukaan tehtäviin päivitetään sprintin aikana arviota siitä kuinka paljon aikaa tehtävän tekeminen vielä vie. [Schwaber and Sutherland, 2013]

Inkrementti (increment) esitettiin kohdassa 4.3. Inkrementti on uusi versio tuotteesta, joka sisältää tuotteen edellisen version lisättynä sprintissä toteutetuilla ominaisuuksilla. Sprintin päättyessä inkrementin täytyy olla ”käytettävä” (useable), jotta uusi versio voidaan julkaista tuotantokäyttöön [Schwaber and Sutherland, 2013]. Inkrementin pitää siis täyttää tuotteelle asetetut laatuvaatimukset.

4.4. Lean

Lean-filosofia on peräisin japanilaisen Toyotan tavasta tuottaa autoja kustannustehokkaasti. Termi *lean* esiteltiin ensimmäisen kerran Womack ja muut [1990] toimesta. Leanin ydinfilosofia perustuu *arvon* (value) tuottamiseen. Perinteisessä tuotantolinjamielessä jokaisen työvaiheen täytyy tuoda lisää arvoa tuotteeseen. Tämän arvontuoton maksimoidakseen Leanissa pyritään poistamaan turha prosessin hukka eli *waste* [Womack et al., 1990].

Kaikki Leanin periaatteet eivät sen tuotantolinjoille kehitetystä filosofiasta johtuen sovi suoraan ohjelmistokehitykseen, sillä ohjelmistokehitys eroaa tavallisesta tuotannosta kahdella tavalla [Measey, 2015]:

1. Teollinen valmistus on toistettavissa oleva prosessi. Ohjelmistokehityksen prosesseja ei voi toistaa, vaan prosessi on jatkuvaa evoluutiota, jossa edellistä versiota tuotteesta muokataan. Ohjelmistokehitysprosessi vaatiikin jatkuvaa oppimista, siitä kuinka toteuttaa ratkaisuja ja kuinka ratkaisujen arvoa voidaan arvioida.
2. Tavallisessa valmistuksessa asiakkaalle toimitetun tuotteen lopputulokseen ei voi juurikaan valmistamisen jälkeen vaikuttaa, mutta ohjelmistoa voidaan muokata käytännössä rajattomasti vielä asiakkaan silmien alla. Leanin periaate turhien asioiden karsimisesta täytyy ohjelmistokehityksessä ajatella tuotteen arvon maksimointina.

Poppendieck ja Poppendieck [2003] kuvaavat Lean-periaatteet ohjelmistokehityksen näkökulmasta. He esittävät seitsemän periaatetta:

1. Optimoi kokonaisuus.
2. Poista turhat vaiheet, jotka eivät tuo lisäarvoa asiakkaalle.
3. Sisällytä laatu tuotteeseen.
4. Opi jatkuvasti.
5. Toimita nopeasti.
6. Sitouta jokainen.
7. Kehity jatkuvasti paremmaksi.

Näiden periaatteiden noudattaminen johtaa Poppendieckien [2003] mukaan laadukkaaseen ohjelmistotuotteeseen. Asiakkaan tarpeisiin optimoitu tuote vastaa parhaiten asiakkaan vaatimuksia ja tuottaa asiakkaalle arvoa ilman turhia ominaisuuksia. Ohjelmiston arvo ei koostu ainoastaan kehitysprosessin aikaansaannoksista, vaan myös ohjelman rakenteesta ja toimitusvalmiudesta.

Suurimman arvon ohjelmistolle tuo kuitenkin sen muokattavuus pitkänkin ajan jälkeen [Poppendieck and Cusumano, 2012].

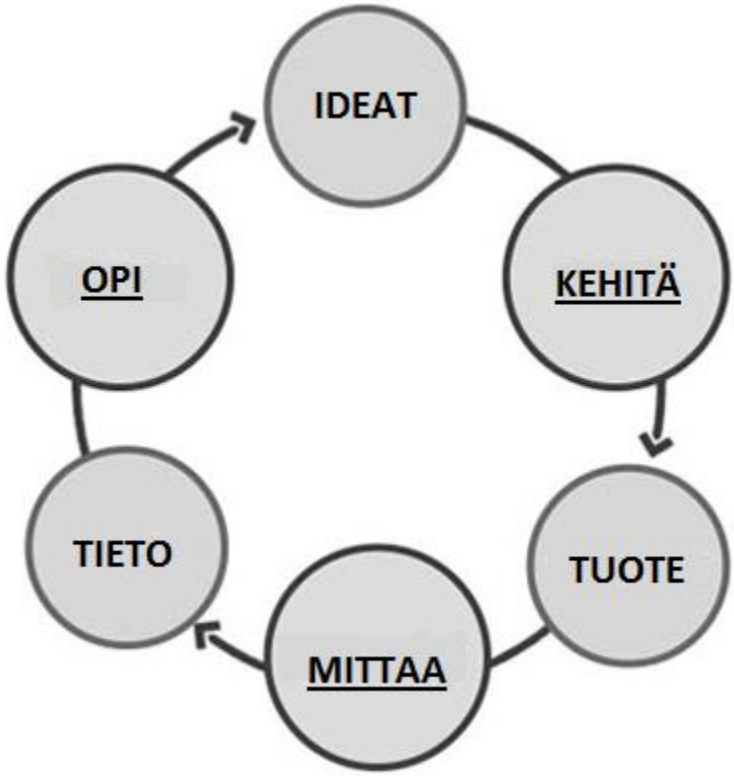
4.5. Kanban

Anderson [2010] esittää, kuinka Kanban-järjestelmää voi käyttää pohjana virtaviivaiselle ohjelmistokehitysprosessille, jossa kehitysprosessi voidaan ajatella työvirtana. Kanbanin pohjana on taulu, jossa on sarakkeisiin ositettuna *arvovirta* (value stream). Taululla hallinnoidaan lappuja, jotka esittävät yhtä tehtävää. Esimerkiksi ohjelmistokehityksessä yksi lappu voi olla uusi toteutettava toiminnallisuus. Lapun sijainti sarakkeissa kertoo, missä arvovirran työvaiheessa kyseinen tehtävä on. Työn edetessä lappu liikkuu taululla vasemmalta oikealle eri työvaiheiden läpi, kunnes se on valmis. Kanbanin ideana on rajoittaa yhden sarakkeen (siis työvaiheen) tehtävien määrää, jotta yhdessä työvaiheessa ei olisi tehtäviä odottamassa. [Anderson, 2010; Poppendieck and Cusumano, 2012]

Kanbanin työvaiheiden kapasiteetin rajoitus pyrkii selkeästi noudattamaan Leanin periaatetta hukan välttämisestä. Visualisoimalla työntilannetta ja tehtävien virtausta Kanbania käyttävä organisaatio voi optimoida kehitysprosessia ja tunnistaa mahdollisia pullonkauloja, joissa esiintyy hukkaa.

4.6. Lean start-up

Lean start-up on lähinnä start-up -henkisten tuotteiden kehittämiseen tarkoitettu liiketoimintamalli, ei niinkään kehitysprosessi. Lean start-up sisältää kuitenkin näkökulmia, joita voidaan hyödyntää myös tavallisissa tuoteorientoituneissa ohjelmistokehitysprojekteissa, vaikka tuote ei olisikaan aivan alkumetreillä, kuten usein start-upien tapauksessa [Ries, 2011].



Kuva 3. Kehitä-mittaa-opi -malli [Ries, 2011; Measey, 2015]

Lean start-up -lähestymistavassa nojaudutaan kehitä-mittaa-opi -malliin (build-measure-learn, kuva 3), jossa aloitteleva yrittäjä jalostaa tuoteideansa kehittämällä siitä ensimmäisen version ja toimittamalla asiakkaalle. Toimituksen jälkeen mitataan tuotteen arvoa asiakkaalle, ja lopuksi kerätystä tiedosta voidaan ammentaa oppia seuraavaa tuotteen kehitysversiota varten [Measey, 2015]. Lähestymistavasta on havaittavissa useita Lean filosofian periaatteita: hukan välttäminen, arvon tuottaminen ja jatkuva oppiminen.

Aluksi tehdään hypoteeseja, ainakin tuotteen arvosta (kuka kokee tuotteen merkitseväksi?) ja potentiaalisesta kasvusta (kuinka asiakkaat löytävät uuden tuotteen?). Hypoteeseja on pystyttävä mittaamaan luotettavasti [Measey, 2015].

Hypoteesien muodostamisen jälkeen Lean-/ketterillä menetelmillä luodaan minimimaalinen tuote (minimum viable product, MVP), joka voidaan toimittaa asiakkaille testiin ja jolla voidaan evaluoida kehitä-mittaa-opi -mallin mukainen silmukka. Asiakkailta kerätään tietoa, minkä jälkeen päätetään muutetaanko hypoteeseja (esimerkiksi vaihdetaan asiakassegmenttiä) vai kehitetäänkö tuotetta eteenpäin [Ries, 2011; Measey, 2015].

4.7. Ketterä ylläpito

Ohjelmiston ylläpito on perinteisesti nähty välttämättömänä ja kalliina käyttöönoton jälkeisenä vaiheena [Harsu, 2003; van Vliet, 2007], joka on erillinen varsinaisesta ohjelmiston kehitysvaiheesta [Sommerville, 2011]. Ohjelmiston ylläpito sisältää useita eri prosesseja ja prosessivaiheita, joiden määritelmiä löytyy kirjallisuudesta laidasta laitaan jo viiden eri vuosikymmenen ajalta. Perinteistä ylläpitoprosessia käsiteltiin luvussa 3.

Ketterät menetelmät on kehitetty ohjelmiston kehitystä, ei ylläpitoa, silmällä pitäen. Kuten todettu, ohjelmiston ylläpidolla ja kehityksellä on eroja. Täten ketterien menetelmien käyttö suoraan ylläpitoprosessissa voi tuottaa huomattavia ongelmia [Heeager & Rose, 2015]. Heeagerin ja Rosen [2015] mukaan ongelmia ketterien menetelmien toimeenpanosta ylläpitoprosesseissa ei ole juurikaan tutkittu.

Kuten Sommerville [2011] toteaa, kehityksen ja ylläpidon selkeä jakaminen erilleen on nykyään epäolennaista. Ohjelmiston kehityskulttuuri on ketterän ideologian myötä kehittynyt enemmän evoluutionaarisempaan suuntaan, jossa ohjelmistoa muokataan koko sen elinkaaren ajan. Vaikka on esitetty lakeja [Lehman, 1980], joiden mukaan jatkuvat muutokset rappeuttavat ohjelmiston rakennetta, tuoreemmat ideologiat kuten XP kuitenkin rohkaisevat jatkuvaan muutokseen. Ketterissä menetelmissä pyritään nopeilla muutoksilla kohti haluttua lopputulosta, mutta samalla myös koodin rakenteen ylläpito (esimerkiksi refaktoroimalla) on ydinosa ketterän kehityksen periaatteita.

Kajko-Mattsson [2008] tuo esiin ketterien ohjelmistoprojektien ylläpitovaiheen ongelmia. Kyselytutkimuksessaan hän keskittyy erityisesti dokumentaation rooliin ketterässä ohjelmistokehityksessä, ja sen vaikutuksiin ohjelmiston ylläpidossa. Useissa kyselyn vastauksissa viitataan suullisen kommunikaation ongelmiin: se vie aikaa ja häiritsee varsinaista työskentelyä. Näitä riskejä olisi kyselyn perusteella mahdollista pienentää dokumentaatiolla sekä tehokkaalla tiedonvaihdolla [Kajko-Mattsson, 2008]. Erityisesti prosessien dokumentaatioon tulisi kiinnittää huomiota, sillä sen avulla voidaan Kajko-Mattssonin [2008] mukaan saavuttaa (1) näkyvyyttä prosessiin, (2) oikeat prosessin mittarit ja sen kautta (3) prosessin korkea kypsyysaste.

Pitkien projektien suuri riski on projektin henkilöstömuutoksista johtuvan hiljaisen järjestelmätiedon katoaminen [Sommerville, 2011]. Tehokkaan tiedonvaihdon edistämiseksi Kajko-Mattsson [2008] esittää ajatuksen *kommunikaatiovastaavasta* (communication officer), jonka tehtävänä olisi levittää projektin osapuolille tietoutta niin kehitysprosessista kuin järjestelmästäkin. Scrumissa esi-

merkiksi Scrum masterin rooliin voidaan ajatella sisältyvän kommunikaatiovastaavan tehtävä.

Sommerville [2011] nostaa esiin kysymyksen siitä, ovatko ketterillä menetelmillä kehitetyt järjestelmät ylläpidettäviä, kun menetelmissä korostetaan minimalistista dokumentaatiota? Sommervillen [2011] kysymyksestä voidaan tulkita mieltymystä vahvaan dokumentaatioon, joka selittynee hänen työhistoriastaan kriittisten järjestelmien parissa. Kriittisten järjestelmien kehittämisessä ja ylläpidossa ajantasainen ja selkeä dokumentaation saatavuus on tärkeä tekijä laadukkaan ja luotettavan järjestelmän kehityksessä. Vaikka Kajko-Mattssonkin [2008] esitti huolensa dokumentaation roolista ketterillä menetelmillä kehitettäessä, vähemmän kriittisissä ja kooltaan pienemmissä järjestelmissä voidaan varmasti keskittyä ketterän filosofian mukaiseen dokumentaatioon, jossa erillistä dokumentaatiota pyritään korvaamaan ymmärrettävällä ja laadukkaalla koodilla.

Dokumentaation tärkeydestä huolimatta kevyemmällä dokumentaatiolla ja muilla ketterien menetelmien sovellutuksilla on saatu nopeutettua kankeita ylläpitoprosesseja (luku 3) sekä edistettyä ohjelmiston laatua [Heeager & Rose, 2015; Ahmad et al., 2016].

Lean-ajattelumallin mukaan hukkaa pyritään välttämään. Tällöin pyritään tehokkaaseen työskentelyyn, jossa eri työvaiheita voidaan suorittaa jouhevasti ilman turhia keskeytyksiä. Kanbanin idea työvaiheiden rajallisista kapasiteeteista on omiaan ketterien ylläpitotehtävien suorittamiseen. Kun kerrallaan suoritettavien tehtävien määrä pidetään rajallisena, voidaan ketterässä projektissa sekä kehittää että ylläpitää tehokkaasti [Concas et al., 2013].

Myös Ahmad ja muut [2016] ovat havainneet, että ylläpitotiimit ovat pystyneet tehostamaan prosessiaan Kanbanin avulla, kun työtä on pystytty visualisoimaan paremmin ja Scrumin tiukat aikarajoitteet ovat poistuneet. Ahmadin ja muiden [2016] tutkimuksessa ylläpitotiimit tunnistivat useita haasteita perinteisen Scrumin käytöstä ylläpitoprosessissa. Esimerkiksi hektisten ylläpitotehtävien sovittaminen sprinttiin tuotti haasteita, sillä asiakkailta tulevat vaatimukset häiritsivät sprintin suorittamista. Lisäksi tehtävien prioriteetit vaihtuivat kesken sprintin, keskeneräisten tehtävien hahmottaminen oli ongelmallista ja kommunikointi muiden projektin osien kanssa vaikeaa.

Kirjallisuudesta on havaittavissa, että ketterässä ylläpidossa pyritään Lean/Kanban -mallien mukaisesti välttämään turhaa hukkaa tehokkaalla ylläpidolla. Näin voidaan tuottaa asiakkaille arvoa nopeasti. Kehityksen ja ylläpidon sekoittuessa voitaneen kysyä, pidetäänkö ylläpitoa nykyisessä ketterien

menetelmien maailmassa yhtä epämukavana kuin perinteinen ylläpidon kirjallisuus antaa olettaa (vrt. kohta 3.6)?

5. Palveluliiketoiminta

Tässä luvussa esitetään lyhyesti palveluliiketoiminnan teoriaa. Nykyisin ohjelmistoja kehitetään enemmän palveluiksi kuin tuotteiksi. Vaikka ohjelmisto olisikin puhdas tuote, sen ympärille myydään useimmiten palveluita. Esimerkiksi ohjelmiston ylläpito voidaan nähdä selkeästi palveluna [van Vliet, 2007]. Onkin tärkeää tunnistaa palvelun elementit, ja kuinka palvelujen avulla luodaan liiketoimintaa.

Perinteisessä liiketoiminnassa tuottaja valmistaa tuotteen ja myy sen asiakkaalle. Kaiken toiminnan keskiössä on tällöin tuote. Nykyisin tuotetaan usein palvelua (service), johon sisältyy perinteisessä mielessä tuote. Tuote on yksi osa palvelunkaltaista *prosessia* (service-like process), jota asiakkaat kuluttavat. Niin kutsutut asiakasrajapinnat (customer interfaces) ovat palvelukeskeisiä: asiakkaaseen ollaan yhteydessä ja pyritään tarjoamaan erilaisia palveluita varsinaisen tuotteen ympärille. [Grönroos, 2006]

Vargo ja Lusch [2008] kuvaavat muutosta tuotekeskeisestä markkinasta palvelukeskeiseksi markkinaksi. Selkeimpänä esimerkkinä tästä on ajatusmallin muutos, missä tuotteen sijaan käydään kauppaa arvolla (value). Arvon tuottamisesta on kyse myös Lean-filosofiassa [Womack et al., 1990]. Muita ajatusmallin muutoksia tuotelähtöisestä ajattelusta palvelulähtöiseen ajatteluun ovat Vargon ja Luschin [2008] mukaan seuraavat:

1. Muutos siitä, että *tehdään* jotain, siihen että *avustetaan asiakasta* luomaan arvoa omissa prosesseissaan.
2. Sen sijaan, että on *tuotettu* ja myyty, onkin *luotu yhdessä.*
3. Asiakas tulisi ymmärtää osana heidän omaa *verkostoaan*, sen sijaan, että heitä ajatellaan vain *eristäytyneinä asiakkaina*.
4. Asiakas tulee ymmärtää *resurssina* eikä *kohteena.*

Palveluliiketoiminta vaatiikin selkeää ajatusmallin muutosta perinteisestä tuotteiden liukuhintavalmistuksesta kohti dynaamisempaa asiakaspalvelua.

Mikä sitten on palvelun ja tuotteen ero? Palvelut ovat abstrakteja, kun taas tuotteet voidaan ajatella enemmän käsin kosketeltavana entiteettinä. Palvelussa on kyse ihmisten välisestä kommunikaatiosta, jolloin palvelun voidaan ajatella olevan heterogeenisempi kuin tavallinen tuote, jota kulutetaan. Täten on myös vaikeampaa määritellä, mistä palvelun laatukokemus muodostuu, kun kyse ei ole mitattavasta asiasta vaan subjektiivisista tuntemuksista. [van Vliet, 2007]

Palvelun ja tuotteen välille ei kuitenkaan voida tehdä selkeää eroa, vaan usein kyseessä on jonkinlainen sekoitus molempia. Esimerkiksi pikaruokaravintoloissa on yhtä tärkeää saada hyvä tuote (ruoka), kuin hyvää palvelua (nopea toimitus). Toisessa esimerkissä matkavakuutuksen (palvelun) lisäksi annetaan matkalaukun osoitelappu (tuote), jolloin asiakkaalle muodostuu käsin kosketeltava mielikuva ostetusta palvelusta. Ohjelmistojen välillä tuote-palvelu -skaalan ääripäissä ovat yksittäinen ohjelmistotuote ja ylläpito. Kun esimerkiksi valmistetaan räätälöityjä ohjelmistoja, ollaan lähellä tuote-palvelu -skaalan puoliväliä. [van Vliet, 2007]

Palvelukeskeisessä liiketoiminnassa palvelun kehitys tapahtuu yhdessä asiakkaan kanssa, joten asiakkaiden tarpeiden tuntemus ja yhteistyö ovat avaimia menestyksekkääseen liiketoimintaan [Tekes, 2016]. Palveluliiketoiminta koostuu kolmesta osa-alueesta [Arantola ja Simonen, 2009]:

1. Asiakasymmärrys.
2. Innovaatiot.
3. Palveluliiketoiminta.

Asiakasymmärryksen voidaan todeta olevan lähtökohta onnistuneelle liiketoiminnalle, sillä kuten todettua, palveluiden tarkoituksena on tuottaa arvoa asiakkaalle. Kun palveluja tarjotaan, on tärkeää ymmärtää millaisia palveluja asiakas arvostaa ja mistä asiakas on valmis maksamaan [Arantola ja Simonen, 2009]. On myös huomattava, että palveluliiketoiminnassa tärkeää ei ole ainoastaan tarjota hyvää palvelua, vaan myös pystyä laskuttamaan luodusta arvosta, jotta liiketoiminnalle syntyy edellytykset.

Ohjelmistoissa asiakkaat arvostavat palvelua. Tämä käy ilmi Stålhanen ja muiden [1997] tutkimuksessa, jossa asiakkailta tiedusteltiin tärkeimpiä ohjelmiston laatutekijöitä. Asiakkaat ovat kaikkein tyytyväisimpiä, kun heidän toiveisiin ja tarpeisiin vastataan nopeasti. Tilastollisesti palvelun laatu oli jopa korkeammalla kuin ohjelmiston toiminnallisuus, mutta tämä ei kuitenkaan suoraan tarkoita toiminnallisuuden olevan asiakkaille toisarvoista. [Stålhane et al., 1997]. Asiakaskokemuksena saattaa olla tärkeämpää virheettömän tuotteen sijaan saada laadukasta palvelua [van Vliet, 2007].

Palveluliiketoiminta mahdollistaa yrityksille uniikit liiketoimintamahdollisuudet. Tavalliset valmistetut tuotteet ovat teoriassa kopioitavissa. Sen sijaan palveluliiketoiminnan aspekteja on vaikeampi kopioida, sillä liiketoiminta perustuu tuotteen lisäksi ihmisten väliseen vuorovaikutukseen. Liiketoiminnan arvo syntyy näiden seikkojen toimivasta kokonaisuudesta. [Lähdeaho, 2010.]

6. Case Lupapiste

Lupapiste on sähköinen asiointipalvelu, jossa asioitsijat voivat hoitaa rakennetun ympäristön lupa- ja ilmoitusasiat. Asiointipalvelun kautta voidaan antaa myös neuvontaa lupa-asioihin liittyen. Lupapistettä käyttävät tavalliset kansalaiset, suunnittelijat, työnjohtajat sekä viranomaiset. Kuntien viranomaiset käsittelevät järjestelmän avulla lupa- ja ilmoitusasiat. Lupapisteen kautta asioitsijat saavat tietoonsa viranhaltijan päätökset sähköisessä muodossa. Lupapiste on yksi valtiovarainministeriön SADe-ohjelman hankkeista. Tutkielman kirjoittaja on työskennellyt ohjelmistokehittäjänä Lupapisteessä lähes kaksi vuotta.

Lupapiste on toteutettu pilvipohjaisena palveluna (Software-as-a-Service), jonka mikä tahansa Suomen kunta voi ottaa käyttöönsä. Palvelua on kehitetty yhteistyössä viranomaisten kanssa. Lupapisteen liiketoiminta perustuu kuukausimaksuun sekä palvelussa tehtävien hakemusten ja ilmoitusten transaktiolaskutukseen. Lisäksi Lupapiste tarjoaa maksullisia lisäpalveluita, joista mainittakoon yrityskäyttäjille suunnattu yritystili sekä tiedonhallinnan kokonaisuus.

Koska Lupapiste on palvelu, sen kehittymistä ohjaa vahvasti käyttäjäpalaute sekä liiketoiminnalliset tarpeet. Arvon tuotto asiakkaille on liiketoiminnan ytimessä. Lupapisteen asiakkaita ovat kaikki sen loppukäyttäjät, mutta viranomaispalveluna kehitystä ohjaa erityisesti kuntien viranomaisten palaute. Kehitysehdotuksia kerätään useiden eri kanavien kautta: kuntatapaamisissa, Yammer-yhteisöissä, tukipalvelun sekä verkkosivuilta löytyvän yhteydenottolomakkeen kautta. Innokkaimmat käyttäjät saattavat ottaa yhteyttä suoraan kehittäjiin. Lisäksi Lupapiste tarjoaa puhelimen ja sähköpostin välityksellä käyttäjätuen, jonka kautta kirjataan ylös paljon kehitysehdotuksia. Saatuja kehitysehdotuksia pohditaan ja priorisoidaan Lupapisteen liiketoiminnan ja tuoteomistajien toimesta.

Tapaustutkimuksessa tutkitaan ylläpidon ja ylläpidettävyyden tilaa Lupapiste-projektissa. Tutkimus jakautuu menetelmällisesti kahtia. Ensin projektin nykytilaa ja toimintatapoja analysoidaan ylläpidon näkökulmasta kuvailevasti. Toiseksi projektin kehittäjiltä ja tuoteomistajilta kerätään tietoa lähdekoodin ja kehitysprosessin tilasta kyselyn avulla sekä analysoidaan saadut vastaukset. Kyselyn tavoitteena on löytää Lupapisteeseen kehityskohteita, joita toimintatapojen analysoinnissa ei löydetty ja saada kokonaiskuva projektin tilasta ylläpitonäkökulmasta. Kysely (liite 1) sisältää sekä väittämiä että avoimia kysymyk-

siä. Kyselyn tuloksia ei liitetä tutkielman osaksi, vaan saatuja vastauksia tulkitaan vapaamuotoisesti.

Tapaustutkimuksen luku jakautuu seuraavasti. Ensin johdatellaan lukijaa Lupapisteen kehitysprosessiin ja ylläpidollisiin näkökulmiin kohdissa 6.1 ja 6.2. Näiden jälkeen paneudutaan kyselyn tuloksiin kohdassa 6.3. Lopuksi, kohdassa 6.4 esitetään johtopäätelminä kehitysehdotuksia tapaustutkimuksen ja ylläpidon teorian avulla.

6.1. Kehitysprosessi

Lupapistettä on kehitetty vuodesta 2012 alkaen. Koko sen elinkaaren ajan Lupapistettä on kehitetty ketterillä menetelmillä. Vuosien aikana kehitysmenetelmiä on jalostettu aina kulloisenkin tarpeen mukaan. Projektin kehittäjien määrä on vaihdellut yhdessä huoneessa työskennelleistä muutamasta ihmisestä yli kymmenen hengen usean paikkakunnan projektiin. Kehitys- ja ylläpitoprosesseja muokataan aina tarpeen mukaan, jotta työskentely olisi mahdollisimman joustavaa mutta tehokasta. Kehitys tapahtuu tällä hetkellä kolmella paikkakunnalla: Tampereella, Helsingissä ja Oulussa.

Lupapistettä kehitetään Scrum-menetelmän periaatteiden mukaisesti. Kehitystä pyritään toteuttamaan Agile-manifestin filosofian mukaisesti koodin laatu ja yksinkertaisuus edellä. Kehityssyklit eli *sprintit* (sprint) ovat kahden viikon mittaisia. Jokaisen sprintin jälkeen palvelusta julkaistaan loppukäyttäjien käyttöön uusi versio. Lupapisteellä on kehitysjono (backlog), jota tuoteomistaja (product owner) yhdessä liiketoiminnan kanssa priorisoivat. Sprintin alussa pidetään sprintin suunnittelupalaveri (sprint planning), jossa tuoteomistaja yhdessä kehitystiimin kanssa käyvät läpi kehitysjonoa yhdessä, ja valitsevat kahden viikon aikana toteutettavat ominaisuudet. Sprintin loppupuolella tuotettu lähdekoodi katselmoidaan. Katselmoinnit toimivat lähdekoodin laadunvarmistuksena. Ennen uuden version julkaisua toteutetut ominaisuudet hyväksymistestataan (acceptance testing) yleensä tuoteomistajien toimesta. Hyväksymistestauksen yhteydessä tehdään päätös siitä viedäänkö toteutettua ominaisuutta tuotantoympäristöön seuraavassa päivityksessä vai lykätäänkö ominaisuuden julkivientiä seuraavan sprintin päätteeksi. Lykkäys voi tapahtua esimerkiksi ominaisuuden keskeneräisyyden tai virheellisen toiminnallisuuden myötä. Sprintin päättyessä pidetään retrospektiivi, jossa tarkastellaan mennyttä sprinttiä. Retrospektiivin jälkeen pidetään seuraavan sprintin suunnittelupalaveri. Jos edellisestä sprintistä on jäänyt tehtäviä kesken, ne otetaan mukaan seuraavalle sprintille. Suurempia muutoksia tarvitsevien ominaisuuskokonaisuuksien ta-

pauksessa muutosten toteuttaminen venyy tietoisesti useampien sprinttien ajalle. Tavoitetila Lupapisteen kehityksessä kuitenkin on, että kokonaisuudet on jaettu pieniksi tehtäviksi, jolloin yksittäiset tehtävät voidaan toteuttaa valmiiksi yhden sprintin, siis kahden viikon, aikana.

Kahden viikon sykli on Lupapisteen kehitystiimin ja liiketoiminnan yhdessä sopima tahti, jolla uusia ominaisuuksia ja korjauksia viedään tuotantoon käyttäjille. Aiemmin uusia versioita on julkaistu sitä mukaa, kun ne ovat valmistuneet. Tästä seurasi, että viikossa saatettiin julkaista jopa neljä uutta versiota. Jotta Lupapisteen kehitys olisi suunnitelmallisempaa ja läpinäkyvämpää, niin sisäisesti kuin myös asiakkaille eli palvelun käyttäjille, on päädytty kahden viikon kehityssykliin. Kehityssykliä voidaan tarpeen vaatiessa muuttaa nopeastikin. Tämä kuitenkin vaatii huolellisen kommunikoinnin kaikkien toimistojen kesken.

Kuten todettu, palveluna Lupapisteen kehittymistä ohjaa vahvasti käyttäjäpalaute sekä liiketoiminnalliset tarpeet. Sprinteillä toteutetaan asiakkaiden ja liiketoiminnan suunnalta tulleita kehitystoiveita. Sprintin ollessa käynnissä palvelun käyttäjiltä saapuu tukipalvelun kautta kehitysehdotuksia ja raportteja mahdollisista virhetilanteista palvelussa. Tuki on yhteydessä usein suoraan kehittäjiin epäselvissä teknistä tukea vaativissa tilanteissa. Jos käyttäjät ovat huomanneet palvelussa jonkin selkeän ongelman, joka täytyy korjata pian, nopea korjaus (hotfix) tuotantojärjestelmään voidaan toteuttaa ohi suunnitellun sprintin. Usein nopeat korjaukset viedään tuotantoon seuraavana yönä, ellei kyseessä ole kriittinen ongelma, jolloin ongelma voidaan korjata tuotantoon erittäin pienellä viiveellä. Nopeita korjauksia varten Lupapisteen kehitystiimistä yksi jäsen toimii sprintin ajan ”sheriffinä”, jonka vastuulla on hoitaa tuen kautta tulevia ongelmia muiden kehittäjien keskittyessä sprintin toteutustehtäviin. Sheriffin tehtävänä on myös reagoida eri ympäristöissä tapahtuviin teknisiin häiriöihin.

6.2. Ylläpidon pohdintaa

Koska Lupapistettä kehitetään palveluliiketoiminnan ehdoilla, kehitysprosessista on verrattain hankalaa erottaa selkeästi, mitkä toimista ovat ylläpitoa ja mitkä uusien ominaisuuksien kehittämistä. Lupapisteessä tapahtuva ylläpito voidaankin määrittää seuraavasti: ylläpito on *säilyttävää* toimintaa eli vanhoja toiminnallisuuksia ja järjestelmän rakennetta vaalitaan samalla, kun uusia toiminnallisuuksia lisätään järjestelmään. Tällä pyritään varmistamaan loppukäyt-

täjän käyttäjäkokemuksen laatu. Laatua pyritään varmistamaan muun muassa koodikatselmoinneilla ja testauksella.

Vaikka ylläpitotoimenpiteitä ei ole Lupapisteen kehityksestä selkeästi erotettavissa, ylläpidolliset toimenpiteet ovat kuitenkin osa jokapäiväistä tekemistä. Uusia ominaisuuksia kehitettäessä saman tehtävän aikana saatetaan tehdä useita eri ylläpidollisia toimenpiteitä. Toki myös korjaavaa ylläpitoa eli bugien korjaamista suoritetaan, jopa yksittäisinä tehtävinä. Korjaavan ylläpidon voidaan ajatella olevan yksi sheriffin toimenkuvaan kuuluvista tehtävistä.

Ehkäisevä ylläpito tähtää ohjelmiston parempaan rakenteeseen ja ylläpidettävyyteen. Kohdassa 2.3 todettiin, että jopa puolet ylläpitoon kuluvasta ajasta kuluu ohjelmiston ymmärtämiseen. Panostus ohjelmiston ylläpidettävyyteen todennäköisesti maksaa itsensä takaisin, kun tulevaisuudessa aikaa lähdekoodin tulkitsemiseen käytetään vähemmän. Tähän kuitenkin liittyy ehkäisevän ylläpidon haasteellisuus: kuinka voidaan liiketoiminnallisesti perustella parannukset lähdekoodin rakenteeseen, kun rakenteen muutos ei ole ulospäin näkyvää, eikä siitä ole suoraa hyötyä loppukäyttäjille, esimerkiksi suorituskykyparannuksina? Kysymys on vaikea. Lupapisteen säilyttävän toiminnan kohdalla ehkäisevästä ylläpidosta voidaan puhua aina, kun uusien ominaisuuksien toteutuksen ohessa suoritetaan ketterien periaatteiden mukaisesti refaktorointia.

Kuten kohdassa 6.1 todettiin, sprinttien loppupuolella suoritetaan lähdekoodin katselmointi. Katselmoinnin järjestäminen on ominaisuuden toteuttaneen kehittäjän vastuulla: hän pyytää yhden tai useamman kollegan tarkastamaan kehittämänsä lähdekoodin. Katselmointikulttuurilla pyritään luomaan kehittäjille positiivinen paine laadukkaan koodin tuottamiseen. Koska toteutetun ominaisuuden lähdekoodi katselmoidaan ennen ominaisuuden hyväksymistä, on kaikkien intresseissä, että tuotettu lähdekoodi on tarpeeksi laadukasta, jotta laatuongelmia ei tarvitse katselmoinnin jälkeen korjailla.

Testauksella varmistetaan, että ohjelmiston osat toimivat halutulla tavalla. Ylläpidon tapauksessa usein esiin nousee regressiotestaus: muutosten toteuttamisen jälkeen järjestelmää testataan uudelleen. Regressiotestauksella pyritään varmistamaan, että järjestelmä toimii myös muutosten jälkeen halutulla tavalla.

Lupapisteessä testejä on neljää eri tyyppiä. Yksikkötestit (unit tests) testaavat yksittäisiä funktioita ja varmistavat niiden oikeellisuuden. Integraatiotestit (integration tests) testaavat järjestelmän palvelinpään toiminnallisuutta testaamalla järjestelmän sisäisten komponenttien väliset toiminnallisuudet (esimerkiksi ketjua http-rajapinta -> liitteen käsittely -> tietokantaan tallennus). Järjestelmätestit (system tests) testaavat Lupapisteen integraatioita ulkoisiin järjes-

telmiin, esimerkiksi kuntien karttapalveluihin. Neljäs testityyppi ovat funktionaaliset testit (funtional tests), jotka testaavat järjestelmän toiminnallisuuden loppukäyttäjän näkökulmasta eli Internet-selaimella. Näitä testejä ajetaan automaattisesti jatkuvan integroinnin periaatteen mukaisesti.

Kun uusia toiminnallisuuksia kehitetään, ja samalla pyritään ylläpitoon eli säilyttävään toimintaan, on nykyisen toiminnallisuuden ymmärtäminen tärkeässä osassa. Ymmärtämiseen liittyvät erityisesti ominaisuudet modulaarisuus ja kompleksisuus (vrt. kohta 2.3). Lupapisteen koodista on havaittavissa osioita, joihin muutosten toteuttaminen on ongelmallista. Esimerkkinä voidaan mainita liitteitä käsittelevä moduuli: liitemoduulin lähdetiedoston rivimäärä on noin 850 riviä, jota voidaan yleisesti pitää melko isona. Lisäksi moduuli on riippuvainen noin 12 erilaisesta apumoduulista (esimerkiksi lokitus, http-kutsut), sekä noin 16 eri sovellusalueen logiikasta vastaavasta moduulista. Liitteiden käsittelyyn liittyvää sovellusalueen logiikkaa ovat esimerkiksi tiedoston tallentamiseen, metadataan, arkistointiin, validointiin ja tulostamiseen liittyvät toiminnallisuudet. Kun suureen liitemoduuliin tehdään muutoksia, muutoksentekijän täytyy huolehtia siitä, että muutos ei riko mitään olemassa olevaa logiikkaa. Mitä enemmän moduulissa on kytköksiä ulospäin, sitä enemmän ylläpitäjä joutuu muistamaan näiden moduulien loogisia suhteita lähdekoodia analysoidessaan. Testien ajaminen ja tulkitseminen voi auttaa ymmärtämään riippuvuuksien syitä ja täten tukea muutosten toteuttamista. Toiminnallisuudet on yleisesti ottaen testattu hyvin, mutta kun jokin testi puuttuu, muutoksesta johtuva heijastusvaikutus saatetaan huomata vasta seuraavan version julkaisun jälkeen, kun loppukäyttäjiltä tulee virheraportteja. Testien olemassaolo ei siis poista huolellisen ymmärtämisen tarvetta.

Kohdassa 4.7 esitettyjä Scrumista johtuvia ongelmia ei Lupapisteessä juurikaan ole havaittavissa. Ensinnäkin palvelua kehitetään palveluorientoituneesti asiakaskunnalle, ei yhdelle asiakkaalle, jolloin asiakkaiden vaatimukset eivät kiilaa suoraan sprintille. Toiseksi sprinttien aikana sheriffi hoitaa erilaiset kiireelliset korjaukset ja ad hoc -tarpeet, jolloin muut kehittäjät voivat keskittyä työskentelemään ennakkoon suunniteltujen sprintin tehtävien kanssa. Kolmanneksi, koska tiimi on kohtuullisen pieni, kommunikaatio toimii hyvin, vaikka tiimi onkin hajautunut fyysisesti useammalle paikkakunnalle.

Palveluorientoituneisuudesta ja sheriffin roolista voidaan tunnistaa Lupapisteen olevan enemmän täysiveristä ohjelmistokehitystä kuin ylläpitoa. Sheriffin rooli voidaan nähdä ylläpitäjänä, joka reagoi mahdollisiin nopeisiin korjaus-

tarpeisiin ja toimii pienempien asioiden selvittäjänä. Tällöin koko muulle tiimille jää resursseja suorittaa kehitystä Scrumin keinoin.

6.3. Kyselyn tulokset

Kysely (liite 1) lähetettiin 12 henkilölle. Kyselyn vastaanottaneista suurin osa oli projektissa tällä hetkellä toimivia kehittäjiä. Lisäksi kysely lähetettiin myös kahdelle tuoteomistajille, joilla on etäinen kosketuspinta lähdekoodiin, mutta näkemystä kehitysprosessiin. Näitä rooleja ei eritellä kyselyssä, eikä siten huomioida kyselyn vastauksia tulkittaessa. Tuoteomistajien vastauksien ei oleteta vääristävän lähdekoodia koskevaa osuutta merkittävästi. Kyselyssä ei kysytty tarkentavia tietoja vastaajista, koska tuloksia tulkitaan vain yleisellä tasolla eikä tarkempia analyysejä tehdä. Kyselyssä oli sekä avoimia, että strukturoituja kysymyksiä. Strukturoidut kyselyt olivat väittämiä, joihin vastattiin neliasteisella Likert-asteikolla, missä arvo 1 on ”täysin eri mieltä” ja arvo 4 on ”täysin samaa mieltä”. Lisäksi joihinkin väittämiin pystyi antamaan tarkennuksen vapaana tekstinä. Mahdollisuutta vastata ”en osaa sanoa” tai ”ei samaa eikä eri mieltä” ei ollut, koska kyselyn tulkitseminen haluttiin pitää yksinkertaisena. Kyselyyn vastauksia tuli yhteensä 11 kappaletta

6.3.1. Lähdekoodi

Kyselyn ensimmäinen osio käsitteli lähdekooditasolla ylläpidettävyyttä. Ylläpidettävyyden ominaisuuksia (kts. taulukko 1) mitattiin väittämien avulla käyttäen Likert-asteikkoa. Kaikki väittämät olivat sävyiltään positiivisia, esimerkiksi ”Lähdekoodia on helppo analysoida”. Avoimina kysymyksinä tiedusteltiin lähdekoodin suhdetta reaalimaailman sovellusalueeseen sekä vapaamuotoista kommenttia ylläpidettävyyteen liittyen.

Ylläpidettävyyden ominaisuuksiin liittyvien väitteiden vastausten keskiarvot liikkuvat asteikon keskivaiheen yläpuolella. Kaikkien vastausten keskiarvo oli 2,6 eli pyöristettynä ylöspäin osittain samaa mieltä. Keskiarvoja tulkittaessa heikoimmaksi ylläpidettävyyden ominaisuudeksi vastaajat kokivat lähdekoodin modulaarisuuden, jonka keskiarvo vastauksissa oli 2,3. Parhaan keskiarvon (3,0) sai lähdekoodin testattavuus. Vastaavasti edellä mainituissa kysymyksissä oli myös suurin (1,0) ja pienin (0,63) keskihajonta. Keskiarvojen tulkitsemisessa tulee noudattaa varovaisuutta, sillä aineiston määrä on melko pieni.

Kyselyssä ylläpidettävyyden ominaisuuksia arvioidessa vastaaja sai myös halutessaan tarkentaa antamaansa vastausta vapaana tekstinä. Näistä kommenteista esiin nousi uudelleenkäytettävyyden vaikea arviointi, sillä kohtaan oli

jätetty useita kommentteja, jotka indikoivat, että vastaajat eivät osanneet arvioida lähdekoodin uudelleenkäytettävyyden tilaa.

Huonoimman keskiarvon saaneeseen modulaarisuuteen otettiin myös kantaa tarkentavissa kommenteissa. Ongelmalliseksi koettiin yksittäisten moduulien suuri koko ja matala koheesio.

Analysoitavuuden puolesta vastauksissa kerrottiin funktionaalisen kielen selkeyttävän analysointia, mutta dynaamisuuden aiheuttavan haasteita. Lisäksi joidenkin funktioiden monimutkaisuus ja kutsuketjujen syvyys aiheuttavat hankaluuksia.

Avoimessa kysymyksessä lähdekoodin suhde reaalimaailmaan koettiin pääosin selkeäksi, vaikka itse sovellusalue koettiin hankalaksi. Lähdekoodissa käytettävän englanninkielisen termistön vastaavuus suomenkieliseen sovellusalueeseen aiheuttaa jonkin verran epäselvyyksiä.

Vastauksissa vapaaseen kommenttiin lähdekoodin ylläpidettävyyteen liittyen moni tunnisti lähdekoodista useamman vuoden kehityksen painolastin, sillä lähdekoodista on tulkittavissa useita erilaisia tyylejä ja ratkaisuja samankaltaisiin ongelmiin. Erityisesti uusimmille projektin jäsenille tämä aiheuttaa epäselvyyksiä siitä, millaisella tyylillä ongelmia tulisi ratkaista. Mutta myös oikeiden arkkitehtuuristen ratkaisujen tekeminen koettiin hankalaksi, koska lähdekoodin eri moduulien vastuunjako ei ole täysin selkeä.

6.3.2. Kehitysprosessi

Kehitysprosessista kyselyssä kysyttiin avoimina kysymyksinä muun muassa dokumentaation tasoa yleisesti, mittauksen ja mittareiden tilaa ja tarpeita, sekä annettiin mahdollisuus antaa vapaamuotoinen kommentti kehitysprosessiin liittyen. Lisäksi Scrum-prosessista kerättiin strukturoidusti tietoa edellä mainitulla Likert-asteikolla. Kyselyssä kysyttiin myös vastaajien mielipidettä siihen, oliko Lupapisteen kehitys vanhan korjaamista, uuden kehittämistä vai molempia. Lisäksi pyydettiin vastaajien mielipidettä siihen, panostetaanko teknisen velan maksuun tarpeeksi. Tässä vastausvaihtoehdot olivat ”sopivasti” (0), ”liian vähän” (1) ja ”liikaa” (2).

Lupapisteen kehitys koettiin vastaajien mielestä kaikkea muuta kuin vain vanhan korjaamiseksi, sillä kukaan ei kokenut Lupapisteen kehitystä pelkästään vanhan toiminnallisuuden korjaamiseksi. Neljä yhdestätoista oli sitä mieltä, että Lupapisteen kehitys on uusien toiminnallisuuksien kehittämistä. Loput seitsemän vastaajaa olivat sitä mieltä, että Lupapisteessä tehdään sekä uuden että vanhan kehitystä yhtä paljon.

Scrum-kehitysprosessista asetettiin useita positiivisen sävyn väittämiä. Esimerkkinä väittämästä on ”sprintin tehtävien aikataulutus on helppoa”. Kaikkien vastausten keskiarvo oli 2,9 eli pyöristettynä ylöspäin osittain samaa mieltä. Huonoimman keskiarvon (2,36) sai sprintin tehtävien aikataulutuksen helppous. Parhaan keskiarvon (3,36) sai tiimin tekeillä olevien tehtävien selkeä saatavuus. Pienin keskihajonta (0,5) oli väitteessä sprintin tehtävien aikataulutuksesta. Suurin keskihajonta oli väitteessä ”kommunikointi on helppoa”, jossa keskihajonta oli (1,0). ”Kommunikointi on helppoa” -väitteen keskiarvo oli 3,1, eli kommunikoinnin helppoudesta oltiin osittain samaa mieltä.

Neljä yhdestätoista vastaajasta oli sitä mieltä, että teknisen velan maksuun panostetaan tällä hetkellä sopivasti. Loppujen seitsemän vastaajan mielestä teknisen velan maksuun panostetaan liian vähän. Yksikään ei vastannut, että teknisen velan maksuun panostettaisiin liikaa.

Kyselyssä tiedusteltiin avoimella kysymyksellä dokumentaation riittävyydestä. Vastauksista voidaan tulkita, että dokumentaation saatavuus on tyydyttävällä tasolla. Muutamista vastauksista on tulkittavissa, että dokumentaatiota ei ole riittävästi tarjolla. Sovellusalueen dokumentaatioon toivottiin parannuksia niin tietosisällön kuin saatavuuden osalta. Saatavuuden osalta on parannettavaa, sillä kaikille vastaajille ei ole selvää, millaista dokumentaatiota on saatavilla ja erityisesti mistä sitä voi etsiä. Kehitysprosessin dokumentaatioon ollaan tyytyväisiä. Samoin arkkitehtuuria ja koodaustyylejä koskenut dokumentaatio sai kiitosta.

Vastaajilta kysyttiin avoimena kysymyksenä mielipidettä prosessin mittaamiseen ja pyydettiin antamaan esimerkkejä mittareista, jotka voisivat tuoda kehitysprosessille enemmän lisäarvoa ja/tai läpinäkyvyyttä. Kysymyksen asettelussa olisi ollut parantamisen varaa, sillä vastauksista kävi ilmi, että kysymykseen oli vaikeaa vastata. Vastauksissa on paljon eroa. Osa oli tyytyväisiä nykyiseen mittaustilanteeseen. Osa toivoi mittareita, esimerkiksi työmääräarvioiden paikkansapitävyyttä. Joissakin vastauksissa myös kyseenalaistettiin prosessin mittaamisen tarvetta. Vastauksista voi kokonaisuutena tulkita suurta epävarmuutta sen suhteen, mitä pitäisi mitata ja miten.

Lopuksi kyselyssä pyydettiin vapaata kommenttia kehitysprosessista. Kommenteista on tulkittavissa tyytyväisyyttä prosessiin. Kolmelle toimistolle jakautunut kehitys mainittiin haasteena, kun jokaiselle toimistolle muodostuu kuitenkin omat prosessinsa ja toimintatavat. Eräässä vastauksessa tuli esiin, että Lupapisteessä on tehty ja voitaisiin tehdä entistä enemmän töitä sopivan

tasapainon löytämiseksi muutosten teon tehokkuuden ja prosessin sujuvuuden välille.

6.4. Johtopäätökset ja kehityskohdat

Kohdassa 6.2 Lupapisteen ylläpitoon otettiin uusi, kirjallisuudesta poikkeava näkökulma: Lupapisteen ylläpito on säilyttävää toimintaa, jossa uutta toiminnallisuutta kehitettäessä pidetään huolta olemassa olevasta toiminnallisuudesta. Ylläpito ja ylläpidettävyydestä huolehtiminen kuuluu olennaisena osana Lupapisteen kehitykseen, vaikka ne eivät olekaan tilastollisesti tunnistettavissa. Ohjelmistotuotteen ja -prosessien kehitys on jatkuvaa iterointia ja kehitystä. Lupapisteessä kumpaakin on mahdollista kehittää varsin ketterästi kohti tehokkaampaa kokonaisuutta. Projektin tila tällä hetkellä on kokonaisuudessaan varsin hyvä. Toki kehityskohtia löytyy. Erityisesti ylläpidettävyyteen tulisi jatkossa kiinnittää entistä enemmän huomiota, kun ohjelmisto kasvaa jatkuvasti uusien ominaisuuksien myötä. Projektin analyysin ja kyselyn perusteella ehdotuksia kehitettäviksi osiksi esitetään seuraavaksi.

Teoriaan peilaten voidaan todeta, että kehittäjien ymmärrystä järjestelmästä voidaan parantaa dokumentoinnilla [Kajko-Mattsson, 2008; Sommerville, 2011]. Toisaalta Agile-manifestin periaatteiden mukaisesti toimiva ohjelmisto on tärkeämpää kuin kattava dokumentaatio. Lupapisteessä järjestelmän peruselementit on dokumentoitu lähdekoodin ulkopuolelle wikiin. Lisäksi versionhallinnassa ylläpidetään kehittäjille suunnattua opasta, johon pyritään dokumentoimaan arkkitehtuuriset suuntaviivaukset sekä teknistä taustatietoa. Kyselyn perusteella kehittäjille suunnattua teknistä opasta kiiteltiin. Kyselystä kävi ilmi, että osa projektin ihmisistä toivoi lisää dokumentaatiota ja osa oli tyytyväisiä nykyiseen tilanteeseen. Ketterän filosofian mukaisesti viitattiin myös siihen, että lähdekoodin pitäisi olla itsessään dokumentoivaa. Kyselystä paljastui, että dokumentaatio pitäisi olla helpommin löydettävissä sekä paremmin jäsenneltyä. Dokumentaation saatavuus ja jäsentely onkin selkeä kehityskohta. Kyselyn vastauksiin ja teoriaan peilaten voitaneen sanoa, että järjestelmän peruselementtien ja sovellusalueen lainalaisuuksien tulisi olla dokumentoituna, mieluiten mahdollisimman ymmärrettävässä muodossa. Tällöin uusille projektin jäsenille tarjotaan mahdollisuus ymmärtää järjestelmää, ja erityisesti sovellusaluetta, jo ennen lähdekoodiin tutustumista.

Kehittäjien ymmärrys järjestelmästä on avainasemassa kustannustehokkaan ja luotettavan ylläpito- ja kehitysprosessin varmistamiseksi. Kokemuksen kautta kerätty hiljainen tieto on arvokasta, kun järjestelmään tehdään muutoksia.

Jatkuvista järjestelmässä tapahtuvista muutoksista on kuitenkin tärkeää koko projektin henkilöstön pysyä ajan tasalla. Kyselyssä nousi esiin lähdekoodin analysoitavuuden haaste, sillä lähdekoodista on havaittavissa useita eri koodaustyylejä. Tämä aiheuttaa epätietoisuutta siitä, mitä tyyliä tulisi käyttää. Ymmärryksen parantamiseksi Lupapisteen kehityskulttuuriin voidaan uutena elementtinä lisätä uusien konseptien (esimerkiksi rakenteiden tai tekniikoiden) esittely joko katselmoinnissa tai retrospektiiveissä. Teknisen esittelyn tarkoituksena on jakaa kaikkien kehittäjien tietoisuuteen uusi konsepti. Parhaassa tapauksessa uudet konseptit dokumentoitaisiin versionhallinnassa sijaitsevaan kehittäjien oppaaseen, jossa Lupapisteen tekniset käytännöt olisivat selkeästi koostettuna. Kun dokumentaation ajantasaisuuteen ja selkeyteen panostetaan, motivaatio niiden tutkimiseen kasvaa. Näin parhaassa tapauksessa vaikutetaan välillisesti koodin laatuun. Tällaisen tiedonjaon myötä kehittäjien tietoisuus järjestelmässä piilevistä mahdollisuuksista lisääntyy ja niitä voidaan ottaa jatkossa käyttöön koko kehitystiimin laajuudella. Samalla kun uusia tekniikoita tuodaan mukaan, pirstoutuneita koodaustyylejä olisi hyvä pyrkiä yhtenäistämään, jotta lähdekoodin analysointi helpottuisi.

Kyselyssä ylläpidettävyyden osalta suurin kehitystarve liittyy lähdekoodin modulaarisuuteen. Erityisesti tiedostojen suuri koko ja heikko koheesio nousivat esille. Kun moduuli on kooltaan suuri ja vastuussa useista eri toiminnallisuuksista, sen ymmärtäminen vaikeutuu huomattavasti. Samoin muutosten tekeminen jättimäisiin moduuleihin on vaivalloista, kun riippuvuuksia muihin moduuleihin on liikaa. Kyselyn tuloksesta voidaan päätellä, että moduulien refaktorointiin on suuri tarve. Rakenteiden refaktoroinnin tuomia hyötyjä tulisi arvioida pitkällä tähtäimellä. Suurikokoiset moduulit ovat kankeita muuttumaan, mutta samalla niiden suuruudesta johtuen suurin osa kehittäjistä joutuu niiden kanssa usein tekemisiin. Tällöin moduulien ymmärtämisen haasteet kertautuvat nopeasti. Myös riski muutosten tuomasta heijastusvaikutuksista kasvaa.

Modulaarisuutta voidaan tutkia metriikoiden avulla. Yksinkertainen lähdekooditiedoston rivienmäärämetriikkakin (LoC) paljastaa liian suuret moduulit nopeasti. Myös kytkeytymismetriikoista olisi hyötyä, kun moduulien riippuvuussuhteista saadaan tietoa. Esimerkiksi aiemmin mainitulla code-maat työkalulla voidaan tiedostojen muutoshistoriaa tutkimalla osoittaa eri moduulien välisiä kytkeytymisiä. Code-maat työkalu voidaan ottaa osaksi jatkuvan integroinnin putkea. Tällöin on mahdollista automaattisesti generoida ja visualisoida

lähdekoodissa esiintyviä potentiaalisia ongelmia, joita on ilman vastaavaa työkalua erittäin hankalaa havaita.

Kehitysprosessin tilanne on melko hyvä. Sprinttisykli koetaan sopivan pituiseksi, työn alla olevat tehtävät ovat näkyvissä kaikille ja kommunikointiin ollaan tyytyväisiä, vaikka kolmen toimiston maantieteellinen jakautuminen koettiin hankalaksi. Vaikka prosessin mittaamisesta ei ollut selkeää visiota, Lupapisteessä voitaisiin esimerkiksi arvioida, voisiko DRE (defect removal efficiency) mittarista olla hyötyä julkaistujen versioiden välisen laadunseurantaan. Tukeen tulleet yhteydenotot voisivat toimia jonkinlaisena indikaattorina mahdollisista laatuongelmista. Yhteydenottojen määrä täytyy toki suhteuttaa palvelun käytön aktiivisuuteen (hakemusten määrä).

Edellä mainittiin, että Ahmadin ja muiden [2016] tutkimuksessa kaikki löydetyt ongelmat eivät ole ajankohtaisia Lupapisteessä. Osa tutkimuksessa löydetyistä ongelmista on kuitenkin vakavasti arvioitava mahdollisina riskeinä Lupapisteen kehitys- ja ylläpitotyön sujuvuudelle. Yksi näistä on kyselyssäkin suurimmaksi ongelmaksi todettu sprintin tehtävien arvioinnin vaikeus. Lupapisteessä on huomattu, että yhdelle sprintille valikoituneet tehtävät harvoin tulevat tehtyä optimaalisessa aikataulussa, sillä usein jotkin tehtävät venyvät seuraavallekin sprintille. Selkeää ratkaisua tähän haasteeseen on hankala löytää. Voidaan kuitenkin tehdä oletus, että arvioinnin vaikeudella on yhteys ylläpidettävyyden ominaisuuksiin analysoitavuus, modulaarisuus, muokattavuus, testattavuus ja uudelleenkäytettävyys. Arviointi on vaikeaa, koska lähdekoodia muutettaessa törmätään ongelmiin, joiden syyt piilevät moninaisesti edellä mainittujen ominaisuuksien sisässä. Lupapisteessä on kuitenkin konsensus siitä, että työmääräarviot ovat vain arvioita. Jokaisen intresseissä on kehittää ylläpidettävyyttä niin, että muutosten toteuttaminen ja sitä myötä työmäärien arviointi helpottuisi.

7. Yhteenveto ja loppupäätelmät

Tässä tutkielmassa esiteltiin ohjelmiston ylläpitoa. Teorian puolesta aihetta pohjustettiin ylläpidettävyyden määritelmällä ja tutustumalla perinteisen ylläpitoprosessin eri vaiheisiin. Jotta voidaan ymmärtää ylläpidon tilaa nykypäivän ohjelmistokehityksessä, tutkielmassa tutustuttiin sekä ketterien menetelmien että palveluliiketoiminnan perusteoriaan. Lisäksi suoritettiin tapaustutkimus, jossa arvioitiin kohdeprojektin tilaa ja näkymiä ylläpidettävyysnäkökulmasta. Tapaustutkimuksen tavoitteena oli saada käsitys siitä, millaista ylläpitoa kohdeprojektissa tapahtuu. Erityisen kiinnostavaa oli saada näkökulmaa siihen, miten ylläpito ja ylläpidettävyys otetaan huomioon, kun ohjelmistoa kehitetään ketterästi palveluliiketoiminnan ehdoilla.

Ylläpidettävyys osoittautui tärkeäksi laatutekijäksi ohjelmiston evoluution kannalta. Standardimäärityksen mukaan ylläpidettävyys koostui viidestä eri ohjelmiston ominaisuudesta: analysoitavuus, modulaarisuus, muokattavuus, testattavuus ja uudelleenkäytettävyys. Näistä mikään ei ole suoraan mitattavissa oleva ominaisuus. Tästä syystä itse ylläpidettävyyden arviointi osoittautui hankalaksi. Kävi ilmi, että ylläpidettävyys onkin useimmiten subjektiivinen arvio. Yksi ylläpidettävyyteen läheisesti liittyvä käsite oli ohjelmiston ymmärtäminen. Kirjallisuuden mukaan ohjelmiston ymmärtämistä auttaa edellä mainittujen ylläpidettävyyden ominaisuuksien lisäksi dokumentaatio. Erityisesti korkeatasoinen ymmärrys sovellusalueesta mahdollistaa lähdekoodin tehokkaan analysoinnin.

Ylläpito on perinteisesti nähty vaiheena, joka alkaa varsinaisen kehitysvaiheen jälkeen, kun asiakas on ottanut ohjelmiston käyttöön. Ylläpidon osuus koko ohjelmiston elinkaaren kustannuksista osoittautui merkittäväksi. Prosessimielessä ylläpito on jaettu selkeisiin kategorioihin ylläpitotoimenpiteiden tyypin mukaan aina standardeja myöten. Ylläpidon teoriasta oli aistittavissa taustalla vaikuttavat jäykät vesiputousmallin mukaiset prosessit.

Tutkittaessa ketteriä menetelmiä ja ylläpitoa oli selkeästi havaittavissa, että ketterien periaatteiden saavuttama suosio ohjelmistotuotannossa on häivyttänyt perinteisesti selvää kehityksen ja ylläpidon välistä rajaa. Ylläpitotoimenpiteitä on nykyään vaikea tunnistaa, kun ohjelmistoja kehitetään entistä enemmän palveluina, joissa pyritään tuomaan asiakkaille jatkuvasti uutta arvoa.

Tapaustutkimuksessa huomattiin, että kohdeprojektissa on käytössä lähestulkoon täysiverinen, uuden projektin kaltainen, kehitysprosessi. Ylläpito sai uuden näkökulman säilyttävänä toimintana, jossa ylläpitotoimenpiteet ovat

sisällytettynä ketterään kehitysprosessiin. Tapaustutkimuksessa tehdyn kyselyn perusteella projektin tila näytti varsin hyvältä, vaikkakin ohjelmiston ylläpidettävyys koettiin paikoin heikoksi.

Lopputulemana ylläpidettävyyden ja ylläpidon teorian avulla saatiin käsitys siitä, mitä ohjelmistolle tapahtuu sen käyttöönoton jälkeen. Tapaustutkimuksen kohdeprojektin ylläpitoa ymmärretään nyt paremmin. Lisäksi löydettiin muutamia kehityskohtia, joilla voidaan edistää kohdeprojektin ohjelmiston ylläpidettävyyttä tulevaisuudessa.

Haasteena tälle tutkielmalle olivat tiukka aikataulu ja tutkimusalueen laajuus. Ylläpito ja ylläpidettävyys osoittautuivat ennakoitua laajemmiksi kokonaisuuksiksi. Näistä kokonaisuuksista jäi läpikäymättä useita tärkeitä seikkoja. Ylläpidettävyyden mittaamisen hankaluus yllätti. Kirjallisuudesta löytyi useita yrityksiä löytää malleja ja mittareita, jotka mittaisivat ylläpidettävyyttä, mutta heikoin tuloksin.

Ylläpitoon liittyy valtavat määrät teoriaa viiden vuosikymmenen ajalta. Ylläpidon ja ketterien menetelmien suhteesta on viime vuosina tullut enenevissä määrin tutkimuksia. Aihe kuitenkin tarvitsee lisää tutkimusta erityisesti palveluliiketoiminnan näkökulmasta, kun ohjelmistot muuttuvat yhä enemmän palveluiksi.

Viiteluettelo

Muhammad Ovais Ahmad, Pasi Kuvaja, Markku Oivo and Jouni Markkula. 2016. Transition of software maintenance teams from scrum to kanban. In: *Proc. of the 49th Hawaii International Conference on System Sciences*. IEEE, 5427-5436.

Bente Anda. 2007. Assessing software system maintainability using structural measures and expert assessments. In: *Proc. of the IEEE International Conference on Software Maintenance*, 204-213.

David J. Anderson. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

Alain April and Alain Abran. 2008. *Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley-IEEE.

Heli Arantola ja Kimmo Simonen. 2009. Palvelemisesta palveluliiketoimintaan – asiakasymmärrys palveluliiketoiminnan perustana. Tekes. 257/2009. Haettu 29.5.2016. https://www.tekes.fi/globalassets/julkaisut/palvelemisesta_palveluliiketoimintaan.pdf

Kent Beck. 1999. *Extreme Programming Explained*: *Embrace Change*. Addison-Wesley.

Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunta, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler and Brian Marick. 2001. Manifesto for agile software development. Checked 5.5.2016. http://agilemanifesto.org/

Barry W. Boehm. 1986. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, **11**, 4, 14-24.

Barry W. Boehm, Chris Abts, A. Winson Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer and Bert Steece. 2000. *Software Cost Estimation with COCOMO II*. Prentice-Hall.

Pierre Bourque and Richard E. Fairley (eds.). 2014. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society.

Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution Research and Practice*, **13**, 3-30.

Giulio Concas, Maria Ilaria Lunesu, Michele Marchesi and Hongyu Zhang. 2013. Simulation of software maintenance process, with and without a work-in-progress limit. *Journal of Software Evolution and Process*, **25**, 1225-1248.

Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, **35**, 5, 684-702.

Mandeep K. Chawla and Indu Chhbara. 2015. SQMMA: Software Quality Model for Maintainability Analysis. In: *Proc. of the 8th Annual ACM India Conference*. 9-17.

Sergio Cozzetti Bertoldi de Souza, Nicolas Anquetil and Kathia M. de Oliveira. 2006. Which documentation for software maintenance? *Journal of the Brazilian Computer Society*, **12**, 3, 31-44.

Sasa M. Deklava. 1992. The influence of the information systems development approach on maintenance. *Journal of MIS Quarterly*, **16**, 3, 355-372.

Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Penny Grubb and Armstrong A. Takang. 2003. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Co.

Christian Grönroos. 2006. Adopting a service logic for marketing. *Marketing Theory*, **6**, 3, 317-333.

Maarit Harsu. 2003. *Ohjelmien ylläpito ja uudistaminen*. Talentum.

Lise Tordup Heeager and Jeremy Rose. 2015. Optimising agile development practices for the maintenance operation: nine heuristics. *Empirical Software Engineering*, **20**, 6. 1762-1784.

IEEE Std 982.1. 1988. IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE.

IEEE. 1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE.

IEEE Std 1219. 1998. IEEE Standard for Software Maintenance. IEEE.

ISO/IEC 14764. 2006. Software Engineering – Software Life Cycle Processes – Maintenance. Second Edition. ISO/IEEE. 44 p.

ISO/IEC 12207. 2008. Software Engineering – Software Life Cycle Processes. Second Edition. ISO/IEEE. 138 p.

ISO/IEC 25010. 2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO/IEEE. 34 p.

ISO/IEC 9126. 2001. Software Engineering – Product quality – Part 1: Quality model. ISO/IEEE.

Mira Kajko-Mattsson. 2001. Towards a business maintenance model. In: *Proc. of the IEEE International Conference on Software Maintenance*. 500-509.

Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Journal of Empirical Software Engineering*, **10**, 1, 31-55.

Mira Kajko-Mattsson. 2008. Problems in agile trenches. In: *Proc. of the ESEM'08 Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 111-119.

Mika V. Mäntylä, Jari Vanhanen and Casper Lassenius. 2004. Bad smells – humans as code critics. In: *Proc. of the 20th IEEE International Conference on Software Maintenance*. 399-408.

Ville-Veikko Kalkkila. 2011. *Ohjelmistokoodin laadulliset metriikat ohjelmistoyrityksessä*. Pro gradu -tutkielma. Informaatiotieteiden yksikkö. Tampereen yliopisto.

Stephen H. Kan. 2002. *Metrics and Models in Software Quality Engineering*. Second Edition. Addison-Wesley.

Meir M. Lehman. 1980. Programs, life cycles, and laws of software evolution. In: *Proc. of the IEEE*, **68**, 9, 1060-1076.

Hareton K. N. Leung. 2002. Estimating maintenance effort by analogy. *Journal of Empirical Software Engineering*, **7**, 2, 157-175.

Bennet P. Lientz and E. Burton Swanson. 1980. *Software Maintenance Management*. Addison-Wesley.

Marika Lähdeaho. 2010. *Palveluliiketoiminnan kehittäminen yritysverkostossa*. Diplomityö. Teknis-taloudellinen tiedekunta. Tampereen teknillinen yliopisto.

Robert C. Martin. 2002. *Agile Software Development, Principles, Patterns and Practices*. Pearson.

Peter Measey. 2015. *Agile Foundations: Principles, Practices and Frameworks*. BCS Learning & Development Ltd.

Alan Padula. 1993. Use of a program understanding taxonomy at Hewlett-Packard. In: *Proc. of the IEEE Second Workshop on Program Comprehension.* 66-70.

Nancy Pennington. 1987. Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive Psychology*, **19**, 295-341.

Mary Poppendieck and Tom Poppendieck. 2003. *Lean Software Development*. Addison-Wesley.

Mary Poppendieck and Michael A. Cusumano. 2012. Lean software development: a tutorial. *IEEE Software*, **29**, 5, 26-32.

Roger S. Pressman. 2005. *Software Engineering: A Practitioner's Approach*. 6. painos. McGraw-Hill.

Eric Ries. 2011. *The Lean Startup*. Crown Business.

Jukka Roihuvaara. 2013. *Kehitysriippuvaisen PHP-ohjelmiston päivitettävyyden arviointi*. Pro gradu -tutkielma. Informaatiotieteiden yksikkö. Tampereen yliopisto.

Ken Schwaber. 1995. SCRUM development process. In: Jeff Sutherland, Cory Casanave, Joaquin Miller, Philip Patel and Glenn Hollowell (eds), *Business Object Design and Implementation: OOPSLA'95 Workshop Proceedings.* 117-134.

Ken Schwaber and Mike Beedle. 2001. *Agile Software Development with Scrum*. Pearson.

Ken Schwaber and Jeff Sutherland. 2013. The Scrum Guide. Checked 5.5.2016. http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf. ScrumInc.

Robert C. Seacord, Daniel Plakosh and Grace A. Lewis. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Processes and Business Practices*. Addison-Wesley.

Amal Shargabi, Syed Ahmad Aljunid, Muthukkaruppanan Annamalai, Shuhaida Mohamed Shuhidan. 2015. Program comprehension levels of abstraction for novices. In: *IEEE International Conference on Computer, Communication and Control Technology*. 211-215.

Ian Sommerville. 2006. *Software Engineering*. Eight Edition. Pearson Education.

Ian Sommerville. 2011. *Software Engineering*. Ninth Edition. Pearson Education.

Tor Stålhane, P. C. Borgersen and K. Arnesen. 1997. In search of the customer's quality view. *Journal of Systems Software*, **38**, 85-93.

V. Suma and T. R. Gopalakrishnan Nair. 2009. Defect management strategies in software development. In: Maurizion A. Strangio (ed.), *Recent Advances in Technologies*. InTech.

Jeff Sutherland. 1995. Business object design and implementation workshop. In: *Proc. of the OOPSLA'95*. 170-175.

E. Burton Swanson. 1976. The dimensions of maintenance. In: *Proc. of the 2nd International Conference on Software Engineering*. 492-497.

Hirotaka Takeuchi and Ikujiro Nonaka. 1986. The new new product development game. *Harvard Business Review*, January 1986, **64**, 1, 137-146.

Tekes. 2016. Palveluliiketoiminta – Mitä palvelujen kehittäminen on? Haettu 29.5.2016. http://www.tekes.fi/ohjelmat-ja-palvelut/palveluliiketoiminta/

Adam Tornhill. 2015. *Your Code as a Crime Scene: Use Forensics Techniques to Arrest Defects, Bottlenecks and Bad Design in Your Programs*. Pragmatic Bookshelf.

Stephen L. Vargo and Robert Lusch. 2008. From goods to service(s): divergences and convergences of logics: *Industrial Marketing Management*, **37**, 3, 254-259.

Hans van Vliet. 2007. *Software Engineering: Principles and Practice*. Wiley.

Anneliese von Mayrhauser and A. Marie Vans. 1997. Program understanding behavior during debugging of large scale software. In: *Proc. of the 17th workshop on Empirical Studies of Programmers*. 157-179.

Billy C. Wagey, Bayu Hendradjaya and M. Sukrisno Mardiyanto. 2015. A proposal of software maintainability model using code smell measurement. In: *Proc. of the International Conference on Data and Software Engineering*, 25-30.

James P. Womack, Daniel T. Jones and Daniel Roos. 1990. *The Machine That Changed the World: The Story of Lean Production*. Free Press.

Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects? In: *Proc. of the 28th IEEE International Conference on Software Maintenance*, 305-315.

Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: an empirical study. *Journal of Systems and Software*, **86**, 10, 2639-2653.

Liite 1: Kysely Lupapisteestä

Kysely luotiin Tampereen yliopiston E-lomakkeen versiolla 3.

Lupapisteen kehitys ja ylläpito

Lupapistettä kehitetään vauhdikkaasti. Kyselyn tarkoituksena on saada viitteitä siitä, mitä asioita koetaan hankaliksi a) lähdekooditasolla b) prosessitasolla.

Kyselyn vastauksia käytetään pro gradu tutkielmani tapaustutkimuksen tueksi.

Tutkielman aiheena on Lupapisteen ylläpito ja tarkoituksena on löytää millaisia haasteita kohdataan lähdekooditasolla ja millaisia haasteita varsinaisen prosessin kanssa. Tutkielman tavoitteena on ymmärtää nykyistä tilannetta sekä löytää kehityskohteita.

Vastaukset käsitellään anonyymina. Lisäksi pidän huolta että tutkielmassa tehtävistä päätelmistä ei ole tunnistettavissa yksittäisiä vastaajia. Vastauksia ei lisätä suoraan osaksi tutkielmaa, esimerkiksi liitteenä, vaan vastauksia tulkitaan yleisellä tasolla.

Kysely jakautuu kahteen osaan:

1. Kysymykset lähdekoodiin liittyen

2. Kysymykset kehitysprosessiin liittyen

Kiitos vaivannäöstäsi.

T: Joni

Perustiedot

	alle 6kk	**6-12kk**	**yli vuoden**
Kauanko olet ollut projektissa?			

Lähdekoodi

Onko lähdekoodista helposti ymmärrettävissä suhde reaalimaailman sovellusalueeseen (esim. Rakennusvalvontaan)?

<avoin kysymys>

Ylläpidettävyyden voidaan ajatella koostuvan seuraavista ominaisuuksista:

1. Analysoitavuus (kuinka helppoa lähdekoodia on lukea ja ymmärtää sen merkitys)
2. Muokattavuus (kuinka helppoa lähdekoodiin on tehdä muutoksia)
3. Testattavuus (kuinka vaivatonta on tehdä testejä)
4. Modulaarisuus (lähdekoodi koostuu mahdollisimman itsenäisistä moduuleista, jolloin muutoksia ei tarvitse tehdä useampiin moduuleihin kerralla)
5. Uudelleenkäytettävyys (samoja koodinpätkiä/funktioita voidaan uudelleen käyttää helposti)

Vastaa seuraaviin ylläpidettävyyden väittämiin sen mukaan, kuinka koet ominaisuuden toteutuvan tällä hetkellä Lupapisteessä.

Ylläpidettävyys

	Täysin eri mieltä	**Osittain eri mieltä**	**Osittain samaa mieltä**	**Täysin samaa mieltä**	Tarkennettavaa?
Lähdekoodia on helppo analysoida					
Lähdekoodia on helppo muokata					
Lähdekoodia on helppo testata					
Lähdekoodi koostuu sopivan kokoisista moduuleista					
Lähdekoodin osia on helppo uudelleenkäyttää					
Lähdekoodin dokumentaatio on hyvällä tasolla					

Vapaamuotoinen kommentti lähdekoodin ylläpidettävyydestä:

<avoin kysymys>

Kehitysprosessi

Onko dokumentaatiota saatavilla riittävästi? Esimerkiksi sovellusalueen toiminnallisuuksista, kehitysprosessista.

<avoin kysymys>

Mitataanko prosessia mielestäsi tarpeeksi? Millaiset mittarit voisivat mielestäsi tuoda kehitysprosessille enemmän lisäarvoa/läpinäkyvyyttä?

<avoin kysymys>

Koetko Lupapisteen kehityksen olevan enemmän...	[ ] vanhan korjaamista [ ] uusien toiminnallisuuksien toteuttamista [ ] molempia

Scrum prosessi

	Täysin eri mieltä	**Osittain eri mieltä**	**Osittain samaa mieltä**	**Täysin samaa mieltä**
Kaksi viikkoa on sopiva sprinttisykli				
Sprintin tehtävien aikataulutus on helppoa				
Sprinttien tavoitteet saavutetaan aikataulussa				
Tieto tiimin työn alla olevista tehtävistä on selkeästi saatavilla				
Kommunikointi on helppoa				
Tehtävien koko on sopiva				
Tehtäviä saa työstää sprintin aikana rauhassa				

	sopivasti	**liian vähän**	**liikaa**	Tarkennettavaa?
Teknisen velan maksuun panostetaan mielestäsi...				

Vapaamuotoinen kommentti kehitysprosessista:

<avoin kysymys>